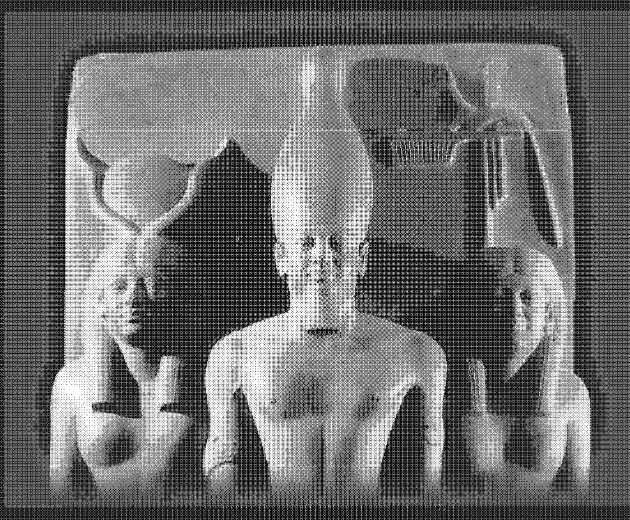
КЛАССИКА COMPUTER SCIENCE

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-Е ИЗДАНИЕ



Э. ТАНЕНБАУМ

PH PTR

ПИТЕР

MODERN OPERATING SYSTEMS

Second Edition

Andrew S. Tanenbaum

PH PTR

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com

КЛАССИКА COMPUTER SCIENCE

Э. ТАНЕНБАУМ

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-Е ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2002

Эндрю Таненбаум

Современные операционные системы

2-е издание

Перевел на русский язык А. Леонтьев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Васильев*
Литературный редактор	*Е. Ваулина*
Художник	*Н. Биржаков*
Иллюстрации	*В. Шендерова*
Корректор	*В. Листова*
Верстка	*Р. Гришанов*

ББК 32.973-018.2
УДК 681.3.066

Таненбаум Э.

Т18 Современные операционные системы. 2-е изд. — СПб.: Питер, 2002. — 1040 с.: ил.

ISBN 5-318-00299-4

Это с нетерпением ожидаемое, переработанное и исправленное издание всемирного бестселлера включает в себя сведения о последних достижениях в области технологий операционных систем. Книга построена на примерах и содержит информацию, необходимую для понимания функционирования современных операционных систем.

Благодаря практическому опыту, приобретенному при разработке нескольких операционных систем, и высокому уровню знания предмета Эндрю Таненбаум смог ясно и увлеченно рассказать о сложных вещах. В книге приводится множество важных подробностей, которых нет ни в одном другом издании.

ISBN 5-318-00299-4
ISBN 0-13-031358-0 (англ.)

ООО «Питер Принт», 196105, Санкт-Петербург, Благодатная ул., д. 67в.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953005 – литература учебная.
Подписано в печать 01.08.02. Формат 70×100 1/16. Усл. п. л. 83,85. Тираж 5000 экз. Заказ № 1039.
Отпечатано с диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Содержание

Сюзанне, Барбаре, Марвину и памяти Брэма и Свити

Об авторе

Эндрю Таненбаум (Andrew S. Tanenbaum) получил степень бакалавра в Массачусетсском технологическом институте и степень доктора философии в Калифорнийском университете в Беркли. Он является профессором кибернетики в университете Врийе (Vrije) в Амстердаме, где возглавляет Группу компьютерных систем. Кроме того, автор является деканом межуниверситетской школы аспирантов по кибернетике и обработке изображений (Advanced School for Computing and Imaging), занимающейся исследованиями в области современных параллельных систем, распределенных систем и систем обработки изображений. Тем не менее он изо всех сил старается не превратиться в бюрократа.

В прошлом автор занимался исследованиями в области компиляторов, операционных систем, компьютерных сетей и локальных распределенных систем. В настоящее время его исследования в основном направлены на разработку глобальных распределенных систем, пользователями которых являются миллионы людей. Результатом этих исследовательских проектов стали более 70 статей в журналах и отчетах конференций. А. Таненбаум является автором пяти книг.

Профессор Таненбаум написал значительное количество программ. Под его руководством разрабатывалась архитектура проекта Amsterdam Compiler Kit — инструмента, широко применяемого для написания портативных компиляторов. Кроме того, он руководил созданием учебной операционной системы MINIX — упрощенной версии системы UNIX, предназначенной для обучения студентов операционным системам. Вместе со своими аспирантами и программистами он помогал в разработке высокопроизводительной распределенной операционной системы Amoeba. В настоящее время системы MINIX и Amoeba свободно распространяются через Интернет и могут использоваться для обучения и исследований.

Его аспиранты, многие из которых получили степень доктора философии, достигли больших успехов. Профессор Таненбаум очень гордится своими учениками. В этом смысле он напоминает курицу-наседку.

Профессор Таненбаум является членом Ассоциации по вычислительной технике ACM (Association for Computing Machinery), старшим членом Института инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronics Engineers), членом Голландской королевской академии искусств и наук. В 1994 году ему была присуждена премия ACM Карла В. Карлстрома (Karl V. Karlstrom) как выдающемуся преподавателю. Э. Таненбаум включен в список *Who's Who in the World*. Его домашняя страница в Интернете расположена по адресу http://www.cs.vu.nl/~ast/.

Предисловие

Мир очень сильно изменился с того момента, когда первое издание этой книги увидело свет в 1992 году. Компьютерные сети и распределенные системы всех типов стали обычным делом. Теперь маленькие дети бродят по Интернету там, где раньше можно было встретить только компьютерных профессионалов. В результате эта книга также претерпела существенные изменения.

Самое заметное изменение заключается в том, что первое издание этой книги было наполовину посвящено однопроцессорным операционным системам и наполовину — распределенным системам. Я выбрал такой формат в 1991 году, потому что в те годы далеко не во всех университетах был специальный курс по распределенным системам, и все, что студенты должны были узнать о распределенных системах, следовало вместить в курс по операционным системам, для которого и предназначалась эта книга. Теперь в большинстве университетов есть отдельный курс по распределенным системам, поэтому необходимости комбинировать эти два предмета в одном курсе и одной книге больше нет. Эта книга предназначается для первого курса по операционным системам и поэтому фокусируется в основном на традиционных однопроцессорных системах.

Я являюсь соавтором двух других книг по операционным системам, и с их учетом возможны два цикла курсов.

Практически ориентированный цикл:

1. Разработка и реализация операционных систем.
2. Распределенные системы.

Традиционный цикл:

1. Современные операционные системы.
2. Распределенные системы.

В первом учебном цикле используется операционная система MINIX и предполагается, что студенты будут экспериментировать с системой MINIX в соответствующей лаборатории, предоставляемой первому курсу. Во втором учебном цикле операционная система MINIX не используется. Вместо нее предоставляются небольшие симуляторы, которые могут использоваться студентами для упражнений во время первого курса с использованием данной книги. Эти симуляторы можно найти на web-странице автора по адресу http://www.cs.vu.nl/~ast/, если щелкнуть мышью по ссылке Software and supplementary material for my books.

Помимо главного изменения, заключающегося в переключении акцента книги на однопроцессорные операционные системы, другие существенные изменения состоят в добавлении целых глав по компьютерной безопасности, мультимедийным

операционным системам и Windows 2000, представляющих собой важные и своевременные вопросы. Кроме того, была добавлена новая уникальная глава по проектированию операционных систем.

Другая новая особенность этой книги состоит в том, что во многие главы теперь добавлены разделы, посвященные исследованиям по теме данной главы. Это сделано с целью познакомить читателя с современными трудами по процессам, управлению памятью и т. д. Разделы содержат многочисленные ссылки на современную исследовательскую литературу для заинтересованных читателей. Кроме того, в главе 13 содержится множество ссылок на учебную литературу.

Наконец, к этой книге было добавлено множество разделов, а многие разделы были серьезно пересмотрены. Это разделы по темам: графические интерфейсы пользователя, мультипроцессорные операционные системы, управление энергопотреблением для переносных компьютеров, надежные системы, вирусы, сетевые терминалы, файловые системы для компакт-дисков, RAID, мягкие таймеры, стабильные хранилища, справедливое планирование и новые алгоритмы замещения страниц. Добавлено множество новых задач и многие старые задачи были пересмотрены. Общее количество задач теперь превышает 450. Сборник задач с решениями может быть предоставлен профессорам, использующим эту книгу на своем курсе. Они могут получить копию книги у своего локального представителя издательства Prentice Hall. Кроме того, было добавлено более 250 новых ссылок на новейшую литературу, чтобы привести книгу в соответствие с современностью.

Несмотря на удаление из книги более чем 400 страниц старого материала, книга увеличилась в размерах благодаря добавлению нового. Книга все еще годится для семестрового курса или курса, состоящего из двух четвертей, но, вероятно, слишком длинна для курса из одной четверти или одного триместра большинства университетов. По этой причине при написании этой книги была предусмотрена ее модульная структура. Любой курс по операционным системам должен включать главы с 1 по 6. Это базовый материал, который должен знать каждый студент.

При наличии дополнительного времени можно изучить дополнительные главы. Каждая из них предполагает, что читатель уже ознакомился с первыми шестью главами, но сами главы с 7 по 12 являются самодостаточными, поэтому может использоваться любое подмножество этих глав и в любом порядке, в зависимости от интересов преподавателя. По мнению автора, главы с 7 по 12 значительно интереснее первых шести глав книги. Преподаватели должны сказать своим студентам, что тем придется съесть капусту, прежде чем они смогут получить двойную порцию торта с фальшивым шоколадом на десерт.

Я хотел бы поблагодарить тех людей, кто оказал мне помощь в пересмотре частей рукописи. Среди них Рида Бацци (Rida Bazzi), Риккардо Беттати (Riccardo Bettati), Фелипе Кабрера (Felipe Cabrera), Ричард Чэпман (Richard Chapman), Джон Коннели (John Connely), Джон Дикинсон (John Dickinson), Джон Элиот (John Elliott), Дебора Фринке (Deborah Frincke), Чандана Гамидж (Chandana Gamage), Роберт Гайст (Robbert Geist), Дэвид Голдс (David Golds), Джим Гриффин (Jim Griffioen), Гари Харкин (Gary Harkin), Франс Кашук (Frans Kaashoek), Муккай Кришнамурти (Mukkai Krishnamoorthy), Моника Лэм (Monica Lam), Джусси Лейво (Jussi Leiwo), Херб Майер (Herb Mayer), Кирк МакКьюзик (Kirk McKusick), Эви Немет (Evi Nemeth), Билл Потвин (Bill Potvin), Прасант Шеной

(Prasant Shenoy), Томас Скиннер (Thomas Skinner), Сиан-Хе Сун (Xian-He Sun), Билльям Терри (William Terry), Роббëрт Ван Ренессе (Robbert Van Renesse) и Маартен ван Стеен (Maarten van Steen). Джеми Ханрахан (Jamie Hanrahan), Марк Русинович (Mark Russinovich) и Дэйв Соломон (Dave Solomon) невероятно много знали о Windows 2000 и очень мне помогли. Особые благодарности следует выразить Элу Вудхаллу (Al Woodhull) за ценные обзоры и мысли о многих новых проблемах, перечисляемых в конце главы.

Мои студенты также очень помогли мне своими комментариями и своей непосредственной реакцией, особенно Стаас де Йонг (Staas de Jong), Ян де Вос (Jan de Vos), Нильс Дрост (Niels Drost), Давид Фоккема (David Fokkema), Ауке Фолькертс (Auke Folkerts), Петер Грюневеген (Peter Groenewegen), Вилько Ибес (Wilco Ibes), Стефан Янсен (Stefan Jansen), Йерун Кетема (Jeroen Ketema), Юри Мулдер (Joeri Mulder), Ирвин Оппенхайм (Irwin Oppenheim), Стеф Пост (Stef Post), Умар Реман (Umar Rehman), Даниель Рийкхоф (Daniel Rijkhof), Маартен Зандер (Maarten Sander), Мориц ван дер Шее (Maurits van der Schee), Рик ван дер Стул (Rik van der Stoel), Марк ван Дрил (Mark van Driel), Деннис ван Вейн (Dennis van Veen) и Томас Зееман (Thomas Zeeman).

Барбара (Barbara) и Марвин (Marvin), как всегда, чудесны, каждый своим неповторимым образом. Наконец, но не в последнюю очередь, я бы хотел поблагодарить Сьюзан (Suzanne) за ее любовь и терпение, не говоря уже обо всех фруктах, которыми она меня потчевала.

Эндрю С. Таненбаум (Andrew S. Tanenbaum)

Глава 1
Введение

Современная компьютерная система состоит из одного или нескольких процессоров, оперативной памяти, дисков, клавиатуры, монитора, принтеров, сетевого интерфейса и других устройств, то есть является сложной комплексной системой. Написание программ, которые следят за всеми компонентами, корректно используют их и при этом работают оптимально, представляет собой крайне трудную задачу. По этой причине компьютеры оснащаются специальным уровнем программного обеспечения, называемым **операционной системой**. Операционная система отвечает за управление всеми перечисленными устройствами и обеспечивает пользователя имеющими простой, доступный интерфейс программами для работы с аппаратурой. Эти системы составляют предмет данной книги.

Расположение операционной системы в общей структуре компьютера показано на рис. 1.1. Внизу находится аппаратное обеспечение, которое во многих случаях само состоит из двух или более уровней (или слоев). Самый нижний уровень содержит физические устройства, состоящие из интегральных микросхем, проводников, источников питания, электронно-лучевых трубок и т. п. То, как они устроены и как работают, относится к сфере деятельности инженеров, специалистов по электронике.

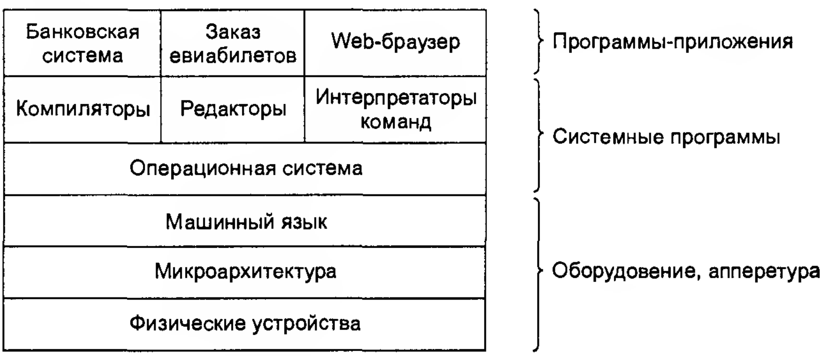


Рис. 1.1. Компьютерная система состоит из аппаратного обеспечения, системных программ и приложений

Выше расположен **микроархитектурный уровень**, на котором физические устройства рассматриваются с точки зрения функциональных единиц. Обычно на этом уровне находятся внутренние регистры центрального процессора (CPU — Central Processing Unit) и арифметико-логическое устройство. На каждом такте

процессора из регистра выбирается один или два операнда, которые обрабатываются в арифметико-логическом устройстве (например, действием операции сложения или логического И). Результат сохраняется в одном или нескольких регистрах. В некоторых машинах операции над данными контролируются программными приложениями, которые называются **микропрограммами**. В других компьютерах такой контроль выполняется напрямую аппаратными цепями.

Определенная система команд передается по маршруту передачи данных. Некоторые команды могут быть выполнены за один цикл передачи данных, другие требуют нескольких циклов. Такие команды могут использовать регистры или другие возможности аппаратуры. Команды, видимые для работающего на ассемблере программиста, формируют уровень **ISA** (Instruction Set Architecture — архитектура системы команд), часто называемый **машинным языком**.

Обычно машинный язык содержит от 50 до 300 команд, служащих преимущественно для перемещения данных по компьютеру, выполнения арифметических операций и сравнения величин. Управление устройствами на этом уровне осуществляется с помощью загрузки определенных величин в специальные **регистры устройств**. Например, диску можно дать команду чтения, записав в его регистры адрес места на диске, адрес в основной памяти, число байтов для чтения и направление действия (чтение или запись). На практике нужно передавать большее количество параметров, а статус операции, возвращаемый диском, достаточно сложен. Кроме того, при программировании многих устройств ввода-вывода (I/O — Input/Output) очень важную роль играют временные соотношения.

Операционная система предназначена для того, чтобы скрыть от пользователя все эти сложности. Она состоит из уровня программного обеспечения, который частично избавляет от необходимости общения с аппаратурой напрямую, вместо этого предоставляя программисту более удобную систему команд. Действие чтения блока из файла в этом случае представляется намного более простым, чем когда нужно заботиться о перемещении головок диска, ждать, пока они установятся на нужное место и т. д.

Над операционной системой на нашем рисунке расположены остальные системные программы. Здесь находятся интерпретатор команд (оболочка), системы окон, компиляторы, редакторы и т. д. Важно понимать, что подобные программы не являются частью операционной системы, хотя обычно поставщики компьютеров устанавливают их на машины. Это очень важное замечание. Под операционной системой обычно понимается то программное обеспечение, которое запускается в **режиме ядра** или, как его еще называют, **режиме супервизора**. Она защищена от вмешательства пользователя с помощью аппаратных средств (мы не рассматриваем в данный момент некоторые старые микропроцессоры, которые вообще не имеют аппаратной защиты). Компиляторы и редакторы запускаются в **пользовательском режиме**. Если пользователю не нравится какой-либо компилятор, он при желании может написать свой собственный, но он не может написать собственный обработчик прерываний системных часов, являющийся частью операционной системы и обычно защищенный аппаратно от попыток его модифицировать.

Существуют системы, в которых это различие размыто. К ним относятся встроенные системы, они могут не иметь режима ядра, или интерпретируемые системы,

подобные основанным на Java операционным системам, в которых для разделения компонентов используется интерпретация, а не оборудование. Но в традиционных компьютерах операционная система представляет собой набор программ, запускающихся в режиме ядра.

Во многих системах есть программы, которые работают в пользовательском режиме, но помогают операционной системе или выполняют специализированные функции. Например, часто встречаются программы, позволяющие пользователям изменять свои пароли. Они не являются частью операционной системы и запускаются не в режиме ядра, но выполняемые ими функции влияют на работу системы, и такие программы должны быть определенным способом защищены от воздействия пользователя.

В некоторых системах части того, что обычно считалось операционной системой (например, файловая система), работают в пространстве пользователя. В таких системах сложно провести четкую границу. Понятно, что все программы, запускающиеся в режиме ядра, является частью операционной системы, но некоторые программы, работающие вне этого режима, могут также относится к операционной системе, или, по крайней мере, тесно с ней связаны.

Наконец, над системными программами расположены прикладные программы. Обычно они покупаются или пишутся пользователем для решения собственных проблем — обработки текста, электронных таблиц, технических расчетов или сохранения информации в базе данных.

Что такое операционная система?

Большинство пользователей компьютеров имеют некоторый опыт общения с операционной системой, но обычно они испытывают затруднения при попытке дать определение операционной системы. В известной степени проблема связана с тем, что операционные системы выполняют две основные, но практически не связанные между собой функции: расширение возможностей машины и управление ее ресурсами. И в зависимости от того, какому пользователю вы зададите вопрос, вы услышите в ответ больше или об одной функции, или о другой. Давайте рассмотрим обе функции.

Операционная система как расширенная машина

Как было упомянуто ранее, **архитектура** (система команд, организация памяти, ввод-вывод данных и структура шин) большинства компьютеров на уровне машинного языка примитивна и неудобна для работы с программами, особенно в процессе ввода-вывода данных. Чтобы это утверждение не показалось голословным, кратко рассмотрим пример того, как происходит ввод-вывод данных с гибкого диска через совместимые микросхемы контроллера NEC PD765, используемые на большинстве персональных компьютеров с процессором Intel. (В этой книге мы будем использовать и термин «гибкий диск», и термин «дискета».) Контроллер PD765 имеет 16 команд, каждая задается передачей от 1 до 9 байт в регистр устройства. Это команды для чтения и записи данных, перемещения головки дис-

ка и форматирования дорожек, а также для инициализации, распознавания, установки в исходное положение и калибровки контроллера и приводов.

Основными командами являются команды read и write (чтение и запись). Каждая из них требует 13 параметров, упакованных в 9 байт. Эти параметры определяют такие элементы, как адрес блока на диске, который нужно прочитать, количество секторов на дорожке, физический режим записи, расстановку промежутков между секторами. Они же сообщают, что делать с меткой адреса данных, которые были удалены. Если вы не можете сразу это осмыслить, не волнуйтесь — полностью это понятно лишь посвященным. Когда выполнение операции завершается, чип контроллера возвращает упакованные в 7 байт 23 параметра, отражающие наличие и типы ошибок. Но этого не достаточно, и программист при работе с гибким диском должен также постоянно знать, включен двигатель или нет. Если двигатель выключен, его следует включить (с длительным ожиданием запуска) прежде, чем данные будут прочитаны или записаны. Двигатель не может оставаться включенным слишком долго, так как гибкий диск изнашивается. Программист вынужден выбирать между длинными задержками во время загрузки и изнашивающимися гибкими дисками (с вероятностью потери данных на них).

Даже если не вдаваться глубже в подробности этого процесса, становится ясно, что обыкновенный программист вряд ли захочет столкнуться с такими деталями при работе с гибким диском (или жестким диском, работа с ним не менее сложна). Вместо этого программисту нужны простые высокоуровневые абстракции. В случае работы с дисками типичной абстракцией является коллекция именованных файлов, содержащихся на диске. Каждый файл может быть открыт для чтения или записи, прочитан или записан, а потом закрыт. А такие детали, как текущее состояние двигателя или использование при записи модифицированной частотной модуляции, не должны содержаться в абстракции, предстающей перед пользователем.

Программа, скрывающая истину об аппаратном обеспечении и представляющая простой список поименованных файлов, которые можно читать и записывать, и является операционной системой. Операционная система не только устраняет необходимость работы непосредственно с дисками и предоставляет простой, ориентированный на работу с файлами интерфейс, но и скрывает множество неприятной работы с прерываниями, счетчиками времени, организацией памяти и другими элементами низкого уровня. В каждом случае абстракция, предлагаемая операционной системой, намного проще и удобнее в обращении, чем то, что может предложить нам непосредственно основное оборудование.

С точки зрения пользователя операционная система выполняет функцию расширенной машины или виртуальной машины, в которой проще программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. То, каким образом операционная система достигает своей цели — долгая история, но мы подробно рассмотрим этот процесс в нашей книге. Подведем итог вышесказанному: операционная система предоставляет нам ряд возможностей, которые могут использовать программы с помощью специальных команд, называемых системными вызовами. Мы приведем примеры наиболее общих системных вызовов далее в этой главе.

Операционная система как менеджер ресурсов

Концепция, рассматривающая операционную систему прежде всего как удобный интерфейс пользователя, — это взгляд сверху вниз. Альтернативный взгляд, снизу вверх, дает представление об операционной системе как о механизме, присутствующем в устройстве компьютера для управления всеми частями этой сложнейшей машины. Современные компьютеры состоят из процессоров, памяти, датчиков времени, дисков, мыши, сетевого интерфейса, принтеров и огромного количества других устройств. В соответствии со вторым подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти и устройств ввода-вывода между различными программами, состязающимися за право их использовать.

Представьте, что случилось бы, если бы на одном компьютере оказались работающими три программы и все они одновременно попытались бы напечатать свои выходные данные на одном и том же принтере. Возможно, первые несколько строк на листе появились бы от первой программы, следующие несколько — из второй программы, затем бы следовало несколько строк от третьей программы и т. д. В результате получилась бы полная неразбериха. Операционная система наводит порядок в подобных ситуациях, буферизируя на диске все данные, предназначенные для печати. В процессе работы программы операционная система сохраняет ее выходные данные на диске во временном файле. Затем, по окончании работы этой программы, система отправляет данные на принтер, в то время как другая программа может продолжать формировать свои выходные данные, не обращая внимания на то, что они пока еще фактически не посылаются на печатающее устройство.

Когда компьютером (или сетью) пользуются несколько пользователей, необходимость в управлении памятью, устройствами ввода-вывода, другими ресурсами и их защите сильно возрастает, поскольку пользователи могут обращаться к ним в абсолютно непредсказуемом порядке. К тому же часто приходится распределять между пользователями не только оборудование, но и информацию (файлы, базы данных и т. д.). С этой точки зрения основная задача операционной системы заключается в отслеживании того, кто и какой ресурс использует, в обработке запросов на ресурсы, в подсчете коэффициента загрузки и разрешении проблем конфликтующих запросов от различных программ и пользователей.

Управление ресурсами включает в себя их мультиплексирование (распределение) двумя способами: во времени и в пространстве. Когда ресурс распределяется во времени, различные пользователи и программы используют его по очереди. Сначала один из них получает доступ к использованию ресурса, потом другой и т. д. Например, несколько программ хотят обратиться к центральному процессору. В этой ситуации операционная система сначала разрешает доступ к процессору одной программе, затем, после того как она поработала достаточное время, другой программе, затем следующей и, в конце концов, опять первой. Определение того, как долго ресурс будет использоваться во времени, кто будет следующим и на какое время ему предоставляется ресурс — это задача операционной системы. Еще один пример временного мультиплексирования — распределение заданий, посылаемых для печати на принтер. Когда задания выстраиваются в очередь для печати на одном принтере, операционной системе каждый раз нужно принимать решение о том, которое из них будет печататься следующим.

Другой вид распределения — это пространственное мультиплексирование. Вместо поочередной работы каждый клиент получает часть ресурса. Обычно оперативная память разделяется между несколькими работающими программами, так что все они одновременно могут постоянно находиться в памяти (например, используя центральный процессор по очереди). Если предположить, что памяти достаточно для того, чтобы хранить несколько программ, эффективнее разместить в памяти сразу несколько программ, чем выделить всю память одной программе, особенно если ей нужна лишь небольшая часть имеющейся памяти. Конечно, при этом возникают проблемы справедливого распределения, защиты памяти и т. д., и для разрешения подобных вопросов существует операционная система. Другой ресурс, распределяемый пространственно, — это диск (жесткий). Во многих системах один диск в одно и то же время может содержать файлы нескольких пользователей. Распределение дискового пространства и отслеживание того, кто какие блоки диска использует, является типичной задачей управления ресурсами, которую также выполняет операционная система.

История операционных систем

История развития операционных систем насчитывает уже много лет. В следующих разделах книги мы кратко рассмотрим некоторые основные моменты этого развития. Так как операционные системы появились и развивались в процессе конструирования компьютеров, то эти события исторически тесно связаны. Поэтому чтобы представить, как выглядели операционные системы, мы обсудим следующие друг за другом поколения компьютеров. Такая схема взаимосвязи поколений операционных систем и компьютеров довольно груба, но она обеспечивает некоторую структуру, без которой ничего не было бы понятно.

Первый настоящий цифровой компьютер был изобретен английским математиком Чарльзом Бэббиджем (Charles Babbage, 1792–1871). Хотя большую часть жизни Бэббидж посвятил попыткам создания своей «аналитической машины», он так и не смог заставить ее работать должным образом. Это была чисто механическая машина, а технологии того времени не были достаточно развиты для изготовления многих деталей и механизмов высокой точности. Не стоит и говорить, что его аналитическая машина не имела операционной системы.

Интересный исторический факт: Бэббидж понимал, что для аналитической машины ему необходимо программное обеспечение, поэтому он нанял молодую женщину по имени Ада Лавлейс (Ada Lovelace), дочь знаменитого британского поэта Лорда Байрона. Она и стала первым в мире программистом, а язык программирования Ada® назван в ее честь.

Первое поколение (1945–55): электронные лампы и коммутационные панели

После неудачных попыток Бэббиджа вплоть до Второй мировой войны в конструировании цифровых компьютеров не было практически никакого прогресса. Примерно в середине 1940-х Говард Айкен (Howard Aiken) в Гарварде, Джон

фон Нейман (John von Neumann) в Институте углубленного изучения в Принстоне, Дж. Преспер Эккерт (J. Presper Eckert), Вильям Мочли (William Mauchley) в Пенсильванском университете, Конрад Цузе (Konrad Zuse) в Германии и многие другие продолжили работу в направлении создания вычислительных машин. На первых машинах использовались механические реле, но они были очень медлительны, длительность такта составляла несколько секунд. Позже реле заменили электронными лампами. Машины получались громоздкими, заполняющими целые комнаты, с десятками тысяч электронных ламп, но все равно они были в миллионы раз медленнее, чем даже самый дешевый современный персональный компьютер.

В те времена каждую машину и разрабатывала, и строила, и программировала, и эксплуатировала, и поддерживала в рабочем состоянии одна команда. Все программирование выполнялось на абсолютном машинном языке, управления основными функциями машины осуществлялось просто при помощи соединения коммутационных панелей проводами. Тогда еще не были известны языки программирования (даже ассемблера не было). Об операционных системах никто и не слышал. Обычный режим работы программиста был таков: записаться на определенное время на специальном стенде, затем спуститься в машинную комнату, вставить свою коммутационную панель в компьютер и провести несколько следующих часов в надежде, что во время работы ни одна из двадцати тысяч электронных ламп не выйдет из строя. Фактически тогда на компьютерах занимались только прямыми числовыми вычислениями, например расчетами таблиц синусов, косинусов и логарифмов.

К началу 50-х, с выпуском перфокарт, установившееся положение несколько улучшилось. Стало возможно вместо использования коммутационных панелей записывать и считывать программы с карт, но во всем остальном процедура вычислений оставалась прежней.

Второе поколение (1955–65): транзисторы и системы пакетной обработки

В середине 50-х изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машина будет работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Машины, теперь называемые мэйнфреймами, располагались в специальных комнатах с кондиционированным воздухом, где ими управлял целый штат профессиональных операторов. Только большие корпорации, правительственные учреждения или университеты могли позволить себе технику, цена которой исчислялась миллионами долларов. Чтобы выполнить задание (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на Фортране или ассемблере), а затем перенести на перфокарты. После этого — принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат.

Когда компьютер заканчивал выполнение какого-либо из текущих заданий, оператор подходил к принтеру, отрывал лист с полученными данными и относил

его в комнату для распечаток, где программист позже мог его забрать. Затем оператор брал одну из колод перфокарт, принесенных из комнаты ввода данных, и считывал их. Если в процессе расчетов был необходим компилятор языка Фортран, то оператору приходилось брать его из картотечного шкафа и загружать в машину отдельно. Из-за одного только хождения операторов по машинному залу впустую терялась масса драгоценного компьютерного времени.

Если учитывать высокую стоимость оборудования, не удивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала **система пакетной обработки.** Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например, IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.

Другие, более дорогостоящие машины, такие как IBM 7094, использовались для настоящих вычислений. Это изображено на рис. 1.2.



а б в г д е

Рис. 1.2. Ранняя система пакетной обработки: программист приносит карты для IBM 1401 (*а*); IBM 1401 записывает пакет заданий на магнитную ленту (*б*); оператор приносит входные данные на ленте к IBM 7094 (*в*); IBM 7094 выполняет вычисления (*г*); оператор переносит ленту с выходными данными на IBM 1401 (*д*); IBM 1401 печатает выходные данные (*е*)

Примерно после часа сбора пакета заданий лента перематывалась, и ее относили в машинную комнату, где устанавливали на лентопротяжном устройстве. Затем оператор загружал специальную программу (прообраз сегодняшней операционной системы), которая считывала первое задание с ленты и запускала его. Выходные данные записывались на вторую ленту вместо того, чтобы идти на печать. Завершив очередное задание, операционная система автоматически считывала с ленты следующее и начинала обрабатывать его. После обработки всего пакета оператор снимал ленты с входной и выходной информацией, ставил новую ленту со следующим заданием, а готовые данные помещал на IBM 1401 для печати в **автономном режиме** (то есть без связи с главным компьютером).

Структура типичного входного задания показана на рис. 1.3. Оно начиналось с карты $JOB, на которой указывалось максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить компилятор языка

Фортран с системной магнитной ленты. Эта карта следовала за программой, которую нужно было компилировать, а после нее шла карта $LOAD, указывающая операционной системе загрузить только что скомпилированную объектную программу. (Скомпилированные программы часто записывались на временных лентах, данные с которых могли стираться сразу после использования, и их загрузка должна была выполняться явно.) Следом шла карта $RUN с данными, дающая операционной системе команду выполнять программу. Наконец, карта завершения $END отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных языков управления и интерпретаторов команд.

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном на них программировали на языке Фортран и ассемблере, а типичными операционными системами были FMS (Fortran Monitor System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

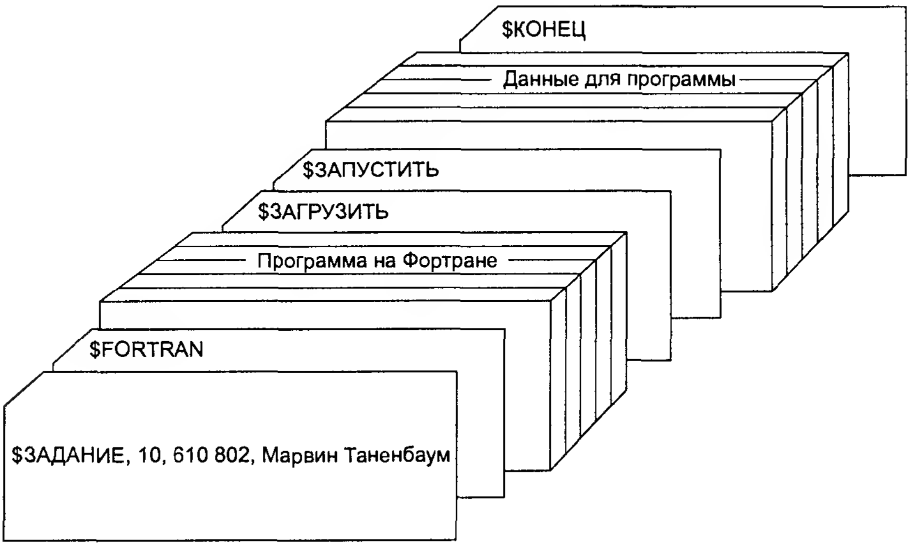


Рис. 1.3. Структура типичного задания FMS

Третье поколение (1965–1980): интегральные схемы и многозадачность

К началу 60-х годов большинство изготовителей компьютеров имело две отдельные, полностью несовместимые производственные линии. С одной стороны, существовали научные крупномасштабные компьютеры с пословной обработкой текста типа IBM 7094, использовавшиеся для числовых вычислений в науке и технике. С другой стороны — коммерческие компьютеры с посимвольной обработкой, такие как IBM 1401, широко используемые банками и страховыми компаниями для сортировки и печатания данных.

Развитие и поддержка двух совершенно разных производственных линий для изготовителей были достаточно дорогим удовольствием. Кроме того, многим покупателям изначально требовалась небольшая машина, однако позже ее возможностей становилось недостаточно и требовался более мощный компьютер, который работал бы с теми же самыми программами, но быстрее.

Фирма IBM попыталась решить эти проблемы разом, выпустив серию машин IBM/360. 360-е были серией программно-совместимых машин, варьирующихся от компьютеров размером с IBM 1401 до машин, значительно более мощных, чем IBM 7094. Эти компьютеры различались только ценой и производительностью (максимальным объемом памяти, быстродействием процессора, количеством разрешенных устройств ввода-вывода и т. д.). Так как все машины имели одинаковую структуру и набор команд, программы, написанные для одного компьютера, могли работать на всех других (по крайней мере, в теории). Кроме того, 360-е были разработаны для поддержки как научных (то есть численных), так и коммерческих вычислений. Одно семейство машин могло удовлетворить нужды всех покупателей. В последующие годы, используя более современные технологии, корпорация IBM выпустила компьютеры, совместимые с 360, эти серии известны под номерами 370, 4300, 3080 и 3090.

360-е стали первой основной линией компьютеров, на которой использовались мелкомасштабные интегральные схемы, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, созданными из отдельных транзисторов. Корпорация IBM добилась мгновенного успеха, а идею семейства совместимых компьютеров скоро приняли и все остальные основные производители. В компьютерных центрах до сих пор можно встретить потомков этих машин. В настоящее время они часто используются для управления огромными базами данных (например, для систем бронирования и продажи билетов на авиалиниях) или как серверы узлов Интернета, которые должны обрабатывать тысячи запросов в секунду.

Основное преимущество «одного семейства» оказалось одновременно и величайшей его слабостью. По замыслу его создателей все программное обеспечение, включая операционную систему OS/360, должно было одинаково хорошо работать на всех моделях компьютеров: и в небольших системах, которые часто заменяли 1401-е и применялись для копирования перфокарт на магнитные ленты, и на огромных системах, заменяющих 7094-е и использовавшихся для расчета прогноза погоды и других сложных вычислений. Кроме того, предполагалось, что одну операционную систему можно будет использовать в системах как с несколькими внешними устройствами, так и с большим их количеством; а также как в коммерческих, так и в научных областях. Но самым важным было, чтобы это семейство машин давало результаты независимо от того, кто и как его использует.

Ни IBM, ни кто-либо еще не мог написать программного обеспечения, удовлетворяющего всем этим противоречивым требованиям. В результате появилась огромная и необычайно сложная операционная система, примерно на два или три порядка превышающая по величине FMS. Она состояла из миллионов строк, написанных на ассемблере тысячами программистов, содержала тысячи и тысячи ошибок, что повлекло за собой непрерывный поток новых версий, в которых пытались исправить эти ошибки. В каждой новой версии устранялась только часть ошибок, вместо них появлялись новые, так что общее их число, вероятно, оставалось постоянным.

Один из разработчиков OS/360, Фред Брукс (Fred Brooks), впоследствии написал остроумную и язвительную книгу с описанием своего опыта работы с OS/360. Мы не можем здесь дать полную оценку этой книги, но достаточно будет сказать, что на ее обложке изображено стадо доисторических животных, увязших в яме с дегтем. Обложка книги [302] демонстрирует похожую точку зрения на операционные системы, бывшие динозаврами в мире компьютеров.

Несмотря на свои огромные размеры и недостатки, OS/360 и подобные ей операционные системы третьего поколения, созданные другими производителями компьютеров, на самом деле достаточно неплохо удовлетворяли требованиям большинства клиентов. Они даже сделали популярными несколько ключевых технических приемов, отсутствовавших в операционных системах второго поколения. Самым важным достижением явилась многозадачность. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании операций ввода-вывода с магнитной ленты или других устройств, центральный процессор просто бездействовал до окончания операции ввода-вывода. При сложных научных вычислениях и ограниченных возможностях процессора устройства ввода-вывода задействовались довольно редко, так что это потраченное впустую время не играло существенной роли. Но при коммерческой обработке данных время ожидания устройства ввода-вывода могло занимать 80 или 90 % всего рабочего времени, поэтому необходимо было что-нибудь сделать во избежание длительного простаивания весьма дорогостоящего процессора.

Решение этой проблемы заключалось в разбиении памяти на несколько частей, называемых разделами, каждому из которых давалось отдельное задание, как показано на рис. 1.4. Пока одно задание ожидало завершения работы устройства ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100 % по времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного любопытства и ущерба со стороны остальных заданий. 360-я и другие системы третьего поколения были снабжены подобными аппаратными средствами.

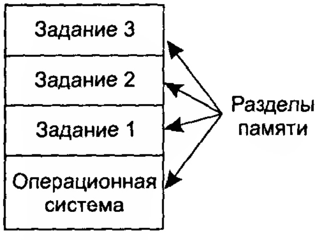


Рис. 1.4. Многозадачная система с тремя заданиями в памяти

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. Всякий раз, когда текущее задание заканчивалось, операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать его. Этот технический прием называется **«подкачкой»** данных

или спулингом (**spooling**, это английское слово произошло от аббревиатуры Simultaneous Peripheral Operation On Line — совместная периферийная операция в интерактивном режиме) и его также используют для выдачи полученных данных. С появлением подкачки стали больше не нужны 1401-е и исчезли многократные переносы магнитных лент.

Хотя операционные системы третьего поколения вполне подходили для больших научных вычислений и справлялись с крупными коммерческими обработками данных, они все еще, по существу, представляли собой разновидности системы пакетной обработки. Многие программисты тосковали по первому поколению машин, когда они могли распоряжаться всей машиной в течение нескольких часов и имели возможность быстро отлаживать свои программы. При системах третьего поколения временной промежуток между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная неуместная запятая могла стать причиной сбоя при компиляции, и получалось, что программист тратил впустую половину дня.

Желание сократить время ожидания ответа привело к разработке **режима разделения времени**, варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Если двадцать пользователей зарегистрированы в системе, работающей в режиме разделения времени, и семнадцать из них думают, беседуют или пьют кофе, то центральный процессор по очереди предоставляется трем пользователям, желающим работать на машине. Так как люди, отлаживая программы, обычно выдают короткие команды (например, компилировать процедуру на пяти страницах[1]) чаще, чем длинные (например, упорядочить файл с миллионами записей), то компьютер может обеспечивать быстрое интерактивное обслуживание нескольких пользователей. При этом он может работать над большими пакетами в фоновом режиме, когда центральный процессор не занят другими заданиями. Первая серьезная система с режимом разделения времени **CTSS** (Compatible Time Sharing System — Совместимая система разделения времени) была разработана в Массачусетсском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094 [75]. Однако режим разделения времени не стал действительно популярным до тех пор, пока не получили широкого распространения необходимые технические средства защиты.

После успеха системы CTSS Массачусетсский технологический институт, система исследовательских лабораторий Bell Labs и корпорация General Electric (тогда главный изготовитель компьютеров) решили начать разработку «компьютерного предприятия общественного пользования» — машины, которая должна была поддерживать сотни одновременных пользователей в режиме разделения времени. Образцом для новой машины послужила система распределения электроэнергии. Когда вам нужна электроэнергия, вы просто вставляете штепсель в розетку и получаете энергии столько, сколько вам нужно. Проектировщики этой системы, известной как **MULTICS** (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), представляли себе одну огромную вычислительную машину, воспользоваться услугой которой мог каждый

---

[1] В этой книге мы будем использовать термины «процедура», «подпрограмма» и «функция» в одном значении. — *Примеч. авт.*

человек в районе Бостона. Мысль о том, что машины, гораздо более мощные, чем их мэйнфрейм GE-645, будут продаваться миллионами по цене тысяча долларов за штуку всего лишь через тридцать лет, казалась чистейшей научной фантастикой, как если бы сегодня кто-либо вздумал проектировать сверхзвуковые трансатлантические подводные поезда.

Успех системы MULTICS был весьма неоднозначен. Эта система разрабатывалась для того, чтобы обеспечить сотни пользователей машиной, немногим более мощной, чем персональный компьютер с процессором Intel 386, хотя при этом имеющей возможность работы со значительно большим количеством устройств ввода-вывода. Это было не так уж безумно, как может показаться, потому что в те дни люди знали, как писать маленькие, эффективные программы — навык, который впоследствии был утерян. Существовало много причин, по которым система MULTICS не захватила весь мир. Не последнюю роль сыграл тот факт, что эта система была написана на языке PL/I, а компилятор языка PL/I появился лишь через несколько лет, к тому же первую версию этого компилятора можно было назвать работоспособной с большой натяжкой. Кроме того, система MULTICS была чрезвычайно претенциозна для своего времени, во многом походя на аналитическую машину Чарльза Бэббиджа в девятнадцатом столетии.

Итак, MULTICS подала много конструктивных идей компьютерным теоретикам, но превратить ее в серьезный продукт и добиться коммерческого успеха оказалось намного тяжелее, чем ожидалось. Система исследовательских лабораторий Bell Labs выбыла из проекта, а компания General Electric совсем оставила компьютерный бизнес. Однако Массачусетсский технологический институт проявил упорство и со временем получил работающую систему. В конце концов, она была продана как коммерческое изделие компанией Honeywell, купившей компьютерный бизнес General Electric, и установлена примерно в восьмидесяти больших компаниях и университетах по всему миру. Несмотря на свою малочисленность, пользователи системы MULTICS были отчаянно преданы ей. Например, компании General Motors, Ford и Управление национальной безопасности США оставили свои системы MULTICS только в конце 90-х годов, через 30 лет после выхода системы.

К настоящему времени идея компьютерного предприятия общественного пользования выдохлась, но она может благополучно вернуться к жизни в форме массивных централизованных Интернет-серверов, выполняющих основную часть работы, к которым будут присоединены относительно «глупые» пользовательские машины. Мотивировка, вероятно, будет следующей: большинство пользователей не захочет администрировать все более усложняющуюся и привередливую систему компьютера и предпочтет доверить эту работу команде профессионалов, работающих на обслуживающую сервер компанию. Электронная коммерция уже развивается в этом направлении, создаются различные компании, управляющие электронными супермаркетами на многопроцессорных серверах, с которыми соединяются простые машины клиентов. Все это очень напоминает замысел системы MULTICS.

Несмотря на неудачу с точки зрения коммерции, система MULTICS значительно повлияла на последующие операционные системы. Это описано в книгах [76, 77, 82, 253, 285]. Системе MULTICS также посвящен все еще активный web-сайт www.multicians.org, с большим количеством информации о системе, ее проектировщиках и пользователях.

Еще одним важным моментом развития во времена третьего поколения был феноменальный рост мини-компьютеров, начиная с выпуска машины PDP-1 корпорацией DEC в 1961 году. Компьютеры PDP-1 обладали оперативной памятью, состоящей всего лишь из 4 К 18-битовых слов, но стоили они по 120 тысяч долларов за штуку (это меньше 5 % от цены IBM 7094) и поэтому расхватывались как горячие пирожки. На некоторых видах нечисловой работы они работали почти с такой же скоростью, как IBM 7094, что дало толчок к появлению новой индустрии. За этой машиной последовала целая серия других PDP (в отличие от семейства IBM, полностью несовместимых), и как кульминация — PDP-11.

Кен Томпсон (Ken Thompson), один из специалистов по компьютерам в Bell Labs, работавший над проектом MULTICS, впоследствии нашел мини-компьютер PDP-7, которым никто не пользовался, и решил написать усеченную однопользовательскую версию системы MULTICS. Эта работа позже развилась в операционную систему **UNIX®**, ставшую популярной в академическом мире, в правительственных управлениях и во многих компаниях.

История развития UNIX уже многократно рассказывалась в самых различных книгах (например [288]). Часть ее будет представлена в главе 10. Пока достаточно сказать, что по причине широкой доступности исходного кода различные организации создали свои собственные (несовместимые) версии, что привело к хаосу. Были разработаны две главные версии: **System V** корпорации AT&T и **BSD** (Berkeley Software Distribution) Калифорнийского университета Беркли. Эти системы, в свою очередь, распадаются на отдельные разновидности. Чтобы стало возможным писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике IEEE разработал стандарт системы UNIX, называемый **POSIX**, который теперь поддерживают большинство версий UNIX. Стандарт POSIX определяет минимальный интерфейс системного вызова, который должны поддерживать совместимые системы UNIX. Некоторые другие операционные системы теперь тоже поддерживают интерфейс POSIX.

Отдельно стоит упомянуть, что в 1987 году автор создал маленький клон системы UNIX для образовательных целей, так называемую систему **MINIX**. Функционально система MINIX очень похожа на UNIX, включая поддержку стандарта POSIX. Существует книга, описывающая внутренние операции MINIX, к которой прилагается листинг исходного кода [326]. Система MINIX свободно распространяется (включая весь исходный код) через Интернет по адресу: www.cs.vu.nl/~ast/minix.html.

Желание иметь свободно распространяемую рабочую (в противоположность образовательной) версию MINIX подвигло финского студента Линуса Торвальдса (Linus Torvalds) к написанию системы **Linux**. Эта система была разработана на основе MINIX и первоначально обладала ее характерными особенностями (например, поддерживала ту же файловую систему). С тех пор система Linux была значительно расширена, но она все еще сохраняет большую часть структуры, общей как для системы MINIX, так и для системы UNIX (на которой и была основана система MINIX). Большая часть того, что будет сказано о UNIX в этой книге, применимо к System V, BSD, MINIX, Linux, а также к другим версиям и клонам UNIX.

Четвертое поколение (с 1980 года по наши дни): персональные компьютеры

Следующий период в эволюции операционных систем связан с появлением Больших Интегральных Схем (LSI, Large Scale Integration) — кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые **микрокомпьютерами**) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно, отличались по цене. Если появление мини-компьютеров позволило отделам компаний и факультетам университетов иметь собственный компьютер, то с появлением микропроцессоров каждый человек получил возможность купить свой собственный персональный компьютер.

В 1974 году, когда компания Intel выпустила Intel 8080 — первый универсальный 8-разрядный центральный процессор, — для него потребовалась операционная система, с помощью которой можно было бы протестировать новинку. Компания Intel привлекла к разработкам и написанию нужной операционной системы одного из своих консультантов Гэри Килдэлла (Gary Kildall). Сначала Килдэлл с другом сконструировали контроллер для 8-дюймового гибкого диска, недавно выпущенного компанией Shugart Associates, и подключили этот диск к процессору Intel 8080. Таким образом, появился первый микрокомпьютер с диском. Затем Килдэлл создал дисковую операционную систему, названную **CP/M** (Control Program for Microcomputers — программа управления для микрокомпьютеров). Когда Килдэлл заявил о своих правах на CP/M, корпорация Intel удовлетворила его просьбу, поскольку не думала, что у микрокомпьютеров с диском есть будущее. Позже Килдэлл создал свою компанию Digital Research для дальнейшего развития и продажи CP/M.

В 1977 году компания Digital Research переработала CP/M, чтобы сделать эту систему пригодной для работы на микрокомпьютерах с процессорами Intel 8080 или Zilog Z80, а также с другими процессорами. Затем было написано множество прикладных программ, работающих в CP/M, что позволило CP/M занимать высшую позицию в мире микрокомпьютеров на протяжении 5 лет.

В начале 80-х корпорация IBM разработала IBM PC (Personal Computer — персональный компьютер) и начала искать для него программное обеспечение. Сотрудники IBM связались с Биллом Гейтсом (Bill Gates), чтобы получить лицензию на право использования его интерпретатора языка Бейсик (BASIC). Они также поинтересовались, не знает ли он операционную систему, которая работала бы на PC. Гейтс посоветовал обратиться к Digital Research, тогда главенствующей компании по операционным системам. Но Килдэлл отказался встречаться с IBM, послав вместо себя подчиненного. Что еще хуже, его адвокат даже отказался подписывать соглашение о неразглашении, касающееся еще не выпущенного PC, чем полностью испортил дело. Корпорация IBM снова обратилась к Гейтсу с просьбой обеспечить ее операционной системой.

После повторного запроса IBM Гейтс выяснил, что у местного изготовителя компьютеров, Seattle Computer Products, есть подходящая операционная система **DOS** (Disk Operating System — дисковая операционная система). Он направился в эту компанию с предложением выкупить DOS (предположительно за $50 000),

которое компания Seattle Computer Products с готовностью приняла. Затем Гейтс создал пакет программ DOS/BASIC, и пакет был куплен IBM. Когда корпорация IBM захотела некоторых усовершенствований в программе, Билл Гейтс пригласил для этой работы Тима Патерсона (Tim Paterson), человека, написавшего DOS, ставшего первым служащим еще не оперившейся компании Гейтса Microsoft. Видоизмененная система была переименована в **MS-DOS** (MicroSoft Disk Operating System) и быстро заняла доминирующее положение на рынке IBM PC. Самым важным оказалось решение Гейтса (как оказалось, чрезвычайно мудрое) продать MS-DOS компьютерным компаниям для установки вместе с их оборудованием, в отличие от попыток Килдэлла продавать CP/M конечным пользователям (по крайней мере, на начальной стадии).

Когда в 1983 году появился компьютер IBM PC/AT с центральным процессором Intel 80286, система MS-DOS уже прочно стояла на ногах, а CP/M доживала свои последние дни. Позже система MS-DOS широко использовалась на компьютерах с процессорами 80386 и 80486. Хотя первоначальная версия MS-DOS была довольно примитивна, последующие версии системы выходили со все лучше разработанными свойствами, включая многое, позаимствованное от UNIX. (Корпорация Microsoft была неплохо информирована о системе UNIX и даже продавала ее микрокомпьютерную версию XENIX в первые годы своего существования.)

CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на вводе команд с клавиатуры. Затем, благодаря исследованиям, проведенным в 60-е годы Дагом Энгельбартом (Doug Engelbart) в научно-исследовательском институте Стэнфорда (Stanford Research Institute), это свойство операционных систем изменилось. Энгельбарт изобрел **графический интерфейс пользователя** (GUI, Graphical User Interface, произносимый как «гуи»[1]), состоящий из окон, значков, различных меню и мыши. Эту идею переняли разработчики из Xerox PARC и встроили в сконструированные ими машины.

Однажды Стив Джобс (Steve Jobs), тот самый, который изобрел компьютер Apple в своем собственном гараже, посетил PARC, где увидел GUI и тотчас осознал его потенциальную ценность, практически не осознаваемую руководством Xerox [307]. Тогда Джобс приступил к созданию Apple с графическим интерфейсом. Это привело к проекту Lisa, который был слишком дорог и потерпел коммерческую неудачу. Вторая попытка Джобса, Apple Macintosh, имела огромный успех не только из-за дешевизны, но и потому, что на нем работал **дружественный интерфейс**, то есть предназначенный для пользователей, ничего не знающих о компьютерах и, более того, вовсе не желающих чему-либо обучаться.

Когда корпорация Microsoft решила создать преемника MS-DOS, она находилась полностью под влиянием успехов компании Macintosh. Была разработана система, получившая название Windows, базой для которой послужил GUI. Система Windows первоначально работала поверх MS-DOS (то есть это была скорее оболочка, чем настоящая операционная система). На протяжении 10 лет, с 1985 по 1995 год, система Windows исполняла роль графической среды поверх MS-DOS. Однако в 1995 году вышла в свет автономная версия Windows 95. Она включила в себя множество особенностей операционной системы MS-DOS, но только для

[1] В переводе означает «липкий». — *Примеч. перев.*

загрузки и выполнения старых программ. В 1998 году была выпущена слегка измененная версия этой системы, получившая название Windows 98. Тем не менее и Windows 95, и Windows 98 все еще содержат большое количество программ 16-разрядного ассемблера Intel.

Другой операционной системой Microsoft стала **Windows NT** (NT означает New Technology — новая технология), которая на определенном уровне совместима с Windows 95, но ее ядро написано полностью заново. Это целиком 32-разрядная система. Дэвид Катлер (David Cutler), главный разработчик Windows NT, был также одним из создателей операционной системы VMS для компьютеров VAX, поэтому некоторые идеи системы VMS присутствуют и в NT. Корпорация Microsoft ожидала, что первая же версия NT вытеснит MS-DOS и все другие версии Windows, так как это была система, намного превосходящая предыдущие, но надежда не оправдалась. И только системе Windows NT 4.0 наконец-то удалось получить относительно широкое распространение, особенно в корпоративных сетях. Версия Windows NT 5.0 была переименована в Windows 2000 в начале 1999 года. Она должна была стать преемником и Windows 98, и Windows NT 4.0. Но этому также не было суждено случиться, поэтому корпорация Microsoft выпустила еще одну версию Windows 98, названную **Windows Me** (Millennium edition — выпуск тысячелетия).

Главным соперником Windows в мире персональных компьютеров становится система UNIX (и ее различные производные). UNIX является самой сильной системой для рабочих станций и других компьютеров старших моделей, таких как сетевые серверы. Она стала особенно популярна на машинах с высокопроизводительными RISC-процессорами (RISC, reduced instruction set computer — компьютер с сокращенным набором команд). На компьютерах с процессорами Pentium популярной альтернативой Windows для студентов и других разнообразных пользователей становится Linux (в дальнейшем мы будем использовать термин «Pentium», подразумевая Pentium I, II, III и 4).

Хотя многие пользователи UNIX, особенно опытные программисты, предпочитают командный интерфейс графическому, почти все UNIX-системы поддерживают оконную систему, созданную в Массачусетсском технологическом институте. Она называется **X Windows**. Эта система оперирует основными функциями окна, позволяя пользователю создавать, удалять, перемещать окна и изменять их размеры с помощью мыши. Часто поверх системы X Windows может быть установлен полный графический интерфейс, например **Motif**, придающий системе UNIX внешний вид системы типа Microsoft Windows или как у компьютера Macintosh.

С середины 80-х годов начали расти и развиваться сети персональных компьютеров, управляемых **сетевыми** и **распределенными операционными системами** [325]. В сетевой операционной системе пользователи знают о существовании многочисленных компьютеров, могут регистрироваться на удаленных машинах и копировать файлы с одной машины на другую. Каждый компьютер работает под управлением локальной операционной системы и имеет своего собственного локального пользователя (или пользователей).

Сетевые операционные системы несущественно отличаются от однопроцессорных операционных систем. Ясно, что они нуждаются в сетевом интерфейсном контроллере и специальном низкоуровневом программном обеспечении, поддерживающем работу контроллера, а также в программах, разрешающих пользователям

удаленную регистрацию в системе и доступ к удаленным файлам. Но эти дополнения, по сути, не изменяют структуры операционной системы.

Распределенная операционная система, напротив, представляется пользователям традиционной однопроцессорной системой, хотя она и составлена из множества процессоров. При этом пользователи не должны беспокоиться о том, где работают их программы или где расположены файлы; все это должно автоматически и эффективно обрабатываться самой операционной системой.

Чтобы создать настоящую распределенную операционную систему, недостаточно просто добавить несколько страниц кода к однопроцессорной операционной системе, так как распределенные и централизованные системы имеют существенные различия. Распределенные системы, например, часто позволяют прикладным задачам одновременно обрабатываться на нескольких процессорах, поэтому требуется более сложный алгоритм загрузки процессоров для оптимизации распараллеливания.

Наличие задержек при передаче данных в сетях означает, что эти алгоритмы должны работать с неполной, устаревшей или даже неправильной информацией. Эта ситуация радикально отличается от однопроцессорной системы, в которой операционная система обладает полной информацией относительно состояния системы.

Онтогенез повторяет филогенез

После опубликования книги Чарльза Дарвина «Происхождение видов» немецкий зоолог Эрнст Хэккель (Ernst Haeckel) сформулировал правило: «Онтогенез повторяет филогенез». Сказав это, он имел в виду, что развитие зародыша (онтогенез) повторяет эволюцию видов (филогенез). Другими словами, человеческая яйцеклетка от момента оплодотворения до того, как стать человеческим ребенком, проходит через состояния рыбы, свиньи и т. д. Современные биологи считают такую модель очень сильно и грубо упрощенной, но все же в ней присутствует зерно истины.

Нечто подобное произошло в компьютерной промышленности. Каждый новый вид (мэйнфрейм, мини-компьютер, персональный компьютер, встроенный компьютер, смарт-карта и т. д.) проходит, видимо, через те же стадии развития, что и их предки. Первые мэйнфреймы программировались полностью на языке ассемблера. Даже такие сложные программы, как компиляторы и операционные системы, в те времена писали на ассемблере. Когда появились мини-компьютеры, на мэйнфреймах уже стали обычными Фортран, Кобол и другие языки программирования высокого уровня, но тем не менее на новых мини-компьютерах программировали на ассемблере (из-за недостатка памяти). Когда были созданы микрокомпьютеры (самые первые персональные компьютеры), программирование на них также велось на ассемблере, несмотря на то, что к этому времени на мини-компьютерах уже применялось программирование на языках высокого уровня. Карманные компьютеры тоже начинались с ассемблерных программ, но очень быстро перешли на языки высокого уровня (в основном за счет того, что к тому моменту они уже разрабатывались на больших машинах). То же самое относится и к смарт-картам.

А теперь взглянем на операционные системы. Первые мэйнфреймы изначально не имели защитного оборудования и поддержки многозадачности, поэтому на них работали простые операционные системы, управляющие в каждый конкретный момент времени только одной загруженной вручную программой. Позже на этих машинах появилось необходимое оборудование и операционные системы, поддерживающие управление одновременно несколькими программами, а затем и полная возможность работы в режиме разделения времени.

Когда мини-компьютеры только появились на свет, на них также не было защитной аппаратуры и в каждый конкретный момент времени могла работать только одна загруженная вручную программа, несмотря на то, что к тому времени многозадачность уже была разработана и хорошо работала в мире мэйнфреймов. Постепенно мини-компьютеры обзавелись защитным оборудованием и появилась возможность одновременной работы на них двух или более программ. На первых микрокомпьютерах также в каждый момент времени могла работать только одна программа, и только позже стал возможным многозадачный режим. Тем же путем развивались карманные компьютеры и смарт-карты.

Диски впервые появились на больших мэйнфреймах и только затем на мини-компьютерах, микрокомпьютерах и т. д. Даже сейчас на смарт-картах нет жесткого диска, но с появлением флэш-памяти вскоре будут созданы эквиваленты дисков и для карт. Лишь после возникновения первых дисков возникли примитивные файловые системы. На компьютере CDC 6600, который смело можно назвать самым мощным мэйнфреймом 60-х годов, пользователи файловой системы имели возможность создавать файл и затем объявлять этот файл постоянным. Это означало, что он останется на диске даже после завершения работы создавшей его программы. Для получения доступа к этому файлу программа должна была подключить его с помощью специальной команды, указав пароль (который задавался в тот момент, когда файл объявлялся постоянным). В сущности, тогда на компьютере был всего один каталог, совместно используемый всеми пользователями. Конфликты имен файлов должны были разрешаться самими пользователями. Так же все начиналось и на мини-компьютерах: ранние файловые системы имели один каталог, общий для всех пользователей; это верно и для ранних микрокомпьютерных файловых систем.

Виртуальная память (то есть виртуальное устройство, позволяющее работать программам, требующим больше памяти, чем физически имеется у компьютера) развивалась точно таким же образом. Сначала она появилась на мэйнфреймах, затем на мини-компьютерах, микрокомпьютерах и постепенно заработала на все меньших и меньших системах. Сети имеют очень похожую историю.

Во всех случаях развитие программного обеспечения диктовалось ростом технологий. Например, на первых микрокомпьютерах было что-то около 4 Кбайт памяти и отсутствовала аппаратная защита. Соответственно, для управления такой крошечной системой не годились языки высокого уровня и многозадачность, они были просто слишком громоздки. По мере того как микрокомпьютеры эволюционировали в современные персональные компьютеры, на них появилось необходимое оборудование, а потом и программное обеспечение для управления этими более сложными устройствами. Вероятно, подобное развитие продолжится в течение последующих лет. В других областях также действует это правило перевоплощения, но в компьютерной промышленности, как нам кажется, развитие происходит заметно быстрее.

Зоопарк операционных систем

Описанное выше развитие компьютеров привело к появлению огромного количества различных операционных систем, далеко не все из которых широко известны. В этом разделе мы кратко рассмотрим семь из них. К некоторым системам из перечисленных ниже мы вернемся позже в нашей книге.

Операционные системы мэйнфреймов

На самом верхнем уровне находятся операционные системы для мэйнфреймов. Эти компьютеры размером с комнату все еще можно встретить в центрах данных больших корпораций. Мэйнфреймы отличаются от персональных компьютеров по своим возможностям ввода-вывода. Довольно часто встречаются мэйнфреймы с тысячью дисков и терабайтами данных, а персональный компьютер с такими параметрами показался бы действительно необычным. Мэйнфреймы как бы возвращаются в виде мощных web-серверов, серверов для крупномасштабных электронно-коммерческих сайтов и серверов для транзакций в бизнесе.

Операционные системы для мэйнфреймов в основном ориентированы на обработку множества одновременных заданий, большинству из которых требуется огромное количество операций ввода-вывода. Обычно они предлагают три вида обслуживания: пакетную обработку, обработку транзакций (групповые операции) и разделение времени. Пакетная обработка представляет собой систему, выполняющую стандартные задания без присутствия пользователей, работающих в интерактивном режиме. Обработка исков в страховых компаниях или составление отчетов о продажах для цепи магазинов – это типичные задания, обрабатываемые в пакетном режиме. Системы обработки транзакций управляют очень большим количеством маленьких запросов, например контролируют процесс работы в банке или бронирование авиабилетов. Каждый отдельный запрос невелик, но система должна отвечать на сотни или тысячи запросов в секунду. Системы, работающие в режиме разделения времени, позволяют множеству удаленных пользователей одновременно выполнять свои задания на одной машине. Хорошим примером является работа с большой базой данных. Все эти функции тесно связаны между собой, и зачастую операционная система мэйнфрейма выполняет их все. Примером операционной системы для мэйнфрейма является OS/390, произошедшая от OS/360.

Серверные операционные системы

Уровнем ниже находятся серверные операционные системы. Они работают на серверах, которые представляют собой или очень большие персональные компьютеры, или рабочие станции, или даже мэйнфреймы. Они одновременно обслуживают множество пользователей и позволяют им делить между собой программные и аппаратные ресурсы. Серверы предоставляют возможность работы с печатающими устройствами, файлами или Интернетом. Интернет-провайдеры обычно запускают в работу несколько серверов для того, чтобы поддерживать одновременный доступ к сети множества клиентов. На серверах хранятся страницы web-

сайтов и обрабатываются входящие запросы. UNIX и Windows 2000 являются типичными серверными операционными системами. Теперь в этих целях стала использоваться и операционная система Linux.

Многопроцессорные операционные системы

Все более часто применяемый способ увеличения мощности компьютеров заключается в соединении нескольких центральных процессоров в одной системе. В зависимости от вида соединения процессоров и разделения работы такие системы называются параллельными компьютерами, мультикомпьютерами или многопроцессорными системами. Для них требуются специальные операционные системы, но зачастую такие операционные системы представляют собой варианты серверных операционных систем со специальными возможностями связи.

Операционные системы для персональных компьютеров

Следующую категорию составляют операционные системы для персональных компьютеров. Их работа заключается в предоставлении удобного интерфейса для одного пользователя. Такие системы широко используются для работы с текстом, электронными таблицами и доступа к Интернету. Наиболее яркие примеры — это Windows 98, Windows 2000, операционная система компьютера Macintosh и Linux. Операционные системы для персональных компьютеров настолько хорошо известны, что вряд ли необходимо представлять здесь их краткий обзор. На самом деле множество людей даже не имеет понятия о существовании других видов операционных систем, кроме той, которой они пользуются.

Операционные системы реального времени

Еще один вид операционной системы — это системы реального времени. Главным параметром таких систем является время. Например, в системах управления производством компьютеры, работающие в режиме реального времени, собирают данные о промышленном процессе и используют их для управления машинами на фабрике. Часто такие процессы должны удовлетворять жестким временным требованиям. Так, если автомобиль передвигается по конвейеру, то каждое действие должно быть осуществлено в строго определенный момент времени. Если сварочный робот сварит шов слишком рано или слишком поздно, то нанесет непоправимый вред машине. Если некоторое действие должно произойти в конкретный момент времени (или внутри заданного диапазона времени), мы имеем дело с **жесткой системой реального времени**.

Существует и другой вид: **гибкая система реального времени**, в которой допустимы случающиеся время от времени пропуски сроков выполнения операции. В эту категорию попадают цифровое аудио и мультимедийные системы. Системы VxWorks и QNX являются хорошо известными операционными системами реального времени.

Встроенные операционные системы

Продолжая двигаться от огромных систем ко все меньшим, мы добрались до «карманных» компьютеров и встроенных систем. Карманный компьютер или **PDA** (Personal Digital Assistant — персональный цифровой помощник) — это маленький компьютер, помещающийся в кармане брюк, выполняющий небольшой набор функций (телефонной записной книжки и блокнота). Встроенные системы, управляющие действиями устройств, работают на машинах, обычно не считающихся компьютерами, например в телевизорах, микроволновых печах и мобильных телефонах. Они часто обладают теми же самыми характеристиками, что и системы реального времени, но при этом имеют особый размер, память и ограничения мощности, что выделяет их в отдельный класс. Примерами таких операционных систем являются PalmOS и Windows CE (Consumer Electronics — бытовая техника).

Операционные системы для смарт-карт

Самые маленькие операционные системы работают на смарт-картах, представляющих собой устройство размером с кредитную карту, содержащее центральный процессор. На такие операционные системы накладываются крайне жесткие ограничения по мощности процессора и памяти. Некоторые из них могут управлять только одной операцией, например электронным платежом, но другие операционные системы на тех же самых смарт-картах выполняют сложные функции. Зачастую они являются патентованными системами.

Некоторые смарт-карты являются Java-ориентированными. Это означает, что ПЗУ (постоянная память, по-английски она называется ROM, Read Only Memory — память только для чтения) смарт-карт содержит интерпретатор виртуальной машины Java (JVM, Java Virtual Machine). Апплеты Java (маленькие программы) загружаются на карту и выполняются JVM-интерпретатором. Некоторые из таких карт могут одновременно управлять несколькими апплетами Java, что приводит к многозадачности и необходимости планирования. Из-за одновременной работы двух и более программ возникает необходимость в управлении ресурсами и защитой. Соответственно, все эти задачи выполняет обычно крайне примитивная операционная система, находящаяся на смарт-карте.

Обзор аппаратного обеспечения компьютера

Операционная система тесно связана с оборудованием компьютера, на котором она должна работать. Аппаратное обеспечение влияет на набор команд операционной системы и управление его ресурсами. Поэтому нам необходим определенный объем знаний о компьютере, по крайней мере нужно представлять, в каком виде оборудование предстает перед программистом.

Концептуально простой персональный компьютер можно представить в виде абстрактной модели, аналогичной той, которая показана на рис. 1.5. Центральный процессор, память и устройства ввода-вывода соединены системной шиной, по

которой они обмениваются друг с другом информацией. Современные персональные компьютеры имеют более сложную структуру, включающую несколько шин; мы вспомним об этом позже. Для начала модели, представленной на рисунке, вполне достаточно. В следующих разделах мы кратко рассмотрим отдельные компоненты и исследуем некоторые аппаратные аспекты, имеющие отношение к разработке операционной системы.

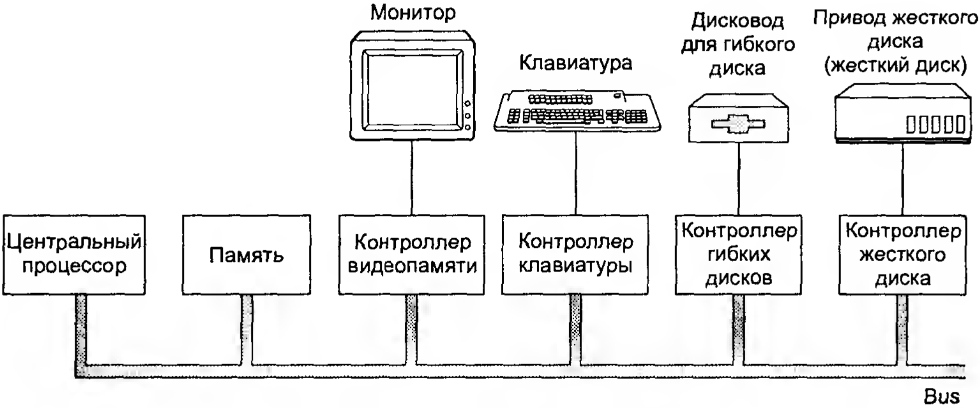


Рис. 1.5. Некоторые компоненты персонального компьютера

Процессоры

«Мозгом» компьютера является **центральный процессор** (CPU — Central Processing Unit). Он выбирает из памяти команды и выполняет их. Обычный цикл работы центрального процессора выглядит так: он читает первую команду из памяти, декодирует ее для определения ее типа и операндов, выполняет команду, затем считывает, декодирует и выполняет последующие команды. Таким образом осуществляется выполнение программ.

Для каждого центрального процессора существует набор команд, который он в состоянии выполнить. Например, процессор Pentium не может обработать программы, написанные для SPARC, а процессор SPARC не может выполнить программы, написанные для Pentium. Поскольку доступ к памяти для получения команд или наборов данных занимает намного больше времени, чем выполнение этих команд, все центральные процессоры содержат внутренние регистры для хранения ключевых переменных и временных результатов. Поэтому набор инструкций обычно содержит команды для загрузки слова из памяти в регистр и сохранения слова из регистра в памяти. Другие команды объединяют два операнда из регистров, памяти или и того и другого и получают результат. Например, складывают два слова и сохраняют результат в регистре или памяти.

Кроме основных регистров, используемых для хранения переменных и временных результатов, большинство компьютеров имеет несколько специальных регистров, видимых для программиста. Один из них называется **счетчиком команд** (PC, program counter), в нем содержится адрес следующей, стоящей в очереди на вы-

полнение команды. После того как команда выбрана из памяти, регистр команд корректируется и указатель переходит к следующей команде.

Еще один регистр процессора называется **указателем стека** (SP, stack pointer). Он содержит адрес вершины стека в памяти. Стек содержит по одному фрейму (области данных) для каждой процедуры, которая уже начала выполняться, но еще не закончена. В стековом фрейме процедуры хранятся ее входные параметры, а также локальные и временные переменные, не хранящиеся в регистрах.

Следующий регистр называется **PSW** (Processor Status Word — слово состояния процессора). Этот регистр содержит биты кода состояний, которые задаются командами сравнения, приоритетом центрального процессора, режимом (пользовательский или режим ядра), и другую служебную информацию. Обычно пользовательские программы могут читать весь регистр PSW целиком, но писать могут только в некоторые из его полей. Регистр PSW играет важную роль в системных вызовах и операциях ввода-вывода.

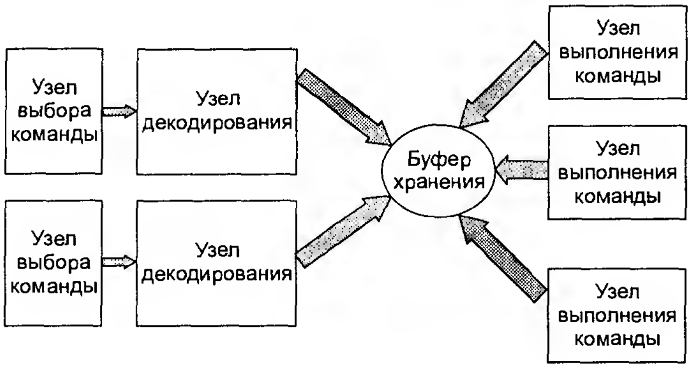
Операционная система должна знать все обо всех регистрах. При временном мультиплексировании центрального процессора операционная система часто останавливает работающую программу для запуска (или перезапуска) другой. Каждый раз при таком прерывании операционная система должна сохранять все регистры процессора, чтобы позже, когда программа продолжит свою работу, их можно было восстановить.

В целях улучшения характеристик центральных процессоров их разработчики давно отказались от простой модели, в которой за один такт может быть считана, декодирована и выполнена только одна команда. Многие современные CPU обладают возможностями выполнения нескольких команд одновременно. Например, у процессора могут быть раздельные модули, занимающиеся выборкой, декодированием и выполнением команд, и во время выполнения команды с номером n он может декодировать команду с номером $n + 1$ и считывать команду с номером $n + 2$. Подобная организация процесса называется **конвейером**, три его стадии продемонстрированы на рис. 1.6, *а*. Часто встречаются и более длинные конвейеры. В большинстве конвейерных конструкций считанная команда должна быть выполнена, даже если в предыдущей команде был принят условный переход. У разработчиков компиляторов и операционных систем это свойство конвейеров часто вызывает головную боль.

Более передовым по сравнению с конвейерной конструкцией является **суперскалярный** центральный процессор, продемонстрированный на рис. 1.6, *б*. В этой структуре присутствует множество выполняющих узлов: один для целочисленных арифметических операций, второй — для операций с плавающей точкой и еще один — для логических операций. За один такт считывается две или более команды, которые декодируются и сбрасываются в буфер хранения, где они ждут своей очереди на выполнение. Когда выполняющее устройство освобождается, оно заглядывает в буфер хранения, интересуясь, есть ли там команда, которую оно может обработать, и если да, то забирает ее и выполняет. В результате команды часто исполняются не в порядке их следования. В большинстве случаев аппаратура должна гарантировать, что результат совпадет с тем, который выдала бы последовательная конструкция. Однако, как мы увидим в дальнейшем, при этом подходе весьма неприятные усложнения коснулись и операционной системы.



а



б

Рис. 1.6. Конвейер с тремя стадиями (*а*); суперскалярный процессор (*б*)

Большинство центральных процессоров, кроме очень простых, используемых во встроенных системах, имеют два режима работы: режим ядра и пользовательский режим. Обычно режим задается битом слова состояния процессора (регистра PSW). Если процессор запущен в режиме ядра, он может выполнять все команды из набора инструкций и использовать все возможности аппаратуры. Операционная система работает в режиме ядра, предоставляя доступ ко всему оборудованию.

В противоположность этому программы пользователей работают в пользовательском режиме, разрешающем выполнение подмножества команд и делающем доступным лишь часть аппаратных средств. Как правило, все команды, включая ввод-вывод данных и защиту памяти, запрещены в пользовательском режиме. Установка бита режима ядра в регистре PSW, естественно, недоступна.

Для связи с операционной системой пользовательская программа должна сформировать **системный вызов**, который обеспечивает переход в режим ядра и активизирует функции операционной системы. Команда TRAP (эмулированное прерывание) переключает режим работы процессора из пользовательского в режим ядра и передает управление операционной системе. После завершения работы управление возвращается к пользовательской программе, к команде, следующей за системным вызовом. Мы рассмотрим в деталях процесс системных вызовов позже в этой главе. В дальнейшем для выделения системных вызовов в тексте мы будем использовать такой же шрифт, как в этом слове: read.

Стоит отметить, что в компьютерах, помимо инструкций для выполнения системных вызовов, есть и другие прерывания. Большинство этих прерываний вызываются аппаратно для предупреждения об исключительных ситуациях, таких как попытка деления на ноль или переполнение при операциях с плавающей точкой. Во всех подобных случаях управление переходит к операционной системе, кото-

рая должна решать, что делать дальше. Иногда нужно завершить программу с сообщением об ошибке. В других случаях ошибку можно проигнорировать (например, при потере значимости числа его можно принять равным нулю). Наконец, если программа объявила заранее, что требуется обработать некоторые виды условий, управление может вернуться назад к программе, позволяя ей самой разрешить появившуюся проблему.

Память

Второй основной составляющей любого компьютера является память. В идеале память должна быть максимально быстрой (быстрее, чем обработка одной инструкции, чтобы работа центрального процессора не замедлялась обращениями к памяти), достаточно большой и чрезвычайно дешевой. На данный момент не существует технологий, удовлетворяющих всем этим требованиям, поэтому используется другой подход. Системы памяти конструируются в виде иерархии слоев, как показано на рис. 1.7.

Верхний слой состоит из внутренних регистров центрального процессора. Они сделаны из того же материала, что и процессор, и так же быстры, как и сам процессор. Поэтому при доступе к ним обычно не возникает задержек. Внутренние регистры предоставляют возможность для хранения 32 × 32 бит на 32-разрядном процессоре и 64 × 64 бит на 64-разрядном процессоре. Это составляет меньше одного килобайта в обоих случаях. Программы сами могут управлять регистрами (то есть решать, что в них хранить) без вмешательства аппаратуры.

В следующем слое находится кэш-память, в основном контролируемая оборудованием. Оперативная память разделена на **кэш-строки**, обычно по 64 байт, с адресацией от 0 до 63 в нулевой строке, от 64 до 127 в первой строке и т. д. Наиболее часто используемые строки кэша хранятся в высокоскоростной **кэш-памяти**, расположенной внутри центрального процессора или очень близко к нему. Когда программа должна прочитать слово из памяти, кэш-микросхема проверяет, есть ли нужная строка в кэше. Если это так, то происходит **результативное обращение к кэш-памяти**, запрос удовлетворяется целиком из кэша и запрос к памяти на шину не выставляется. Удачное обращение к кэшу, как правило, по времени занимает около двух тактов, а неудачное приводит к обращению к памяти с существенной потерей времени. Кэш-память ограничена в размере, что обусловлено ее высокой стоимостью. В некоторых машинах есть два или даже три уровня кэша, причем каждый последующий медленнее и больше предыдущего.

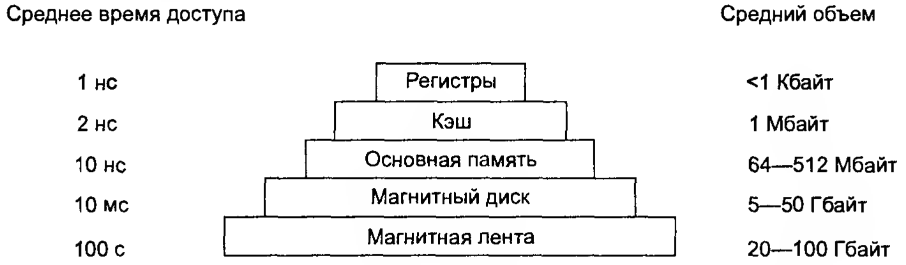


Рис. 1.7. Типичная иерархическая структура памяти. Числа приблизительны

Далее следует оперативная память. Это главная рабочая область запоминающего устройства машины. Оперативную память часто называют **ОЗУ** (оперативное запоминающее устройство, в англоязычной литературе RAM, Random Access Memory — память с произвольным доступом). Раньше иногда ее называли **core memory** — запоминающее устройство на магнитных сердечниках, поскольку в 50-е и 60-е годы в компьютерах для оперативной памяти использовали крошечные намагничиваемые ферритовые сердечники. Сейчас память составляет десятки и сотни мегабайт и растет с потрясающей скоростью. Все запросы центрального процессора, которые не могут быть выполнены кэш-памятью, поступают для обработки в основную память.

Следующим в продемонстрированной на рисунке структуре идет магнитный диск (жесткий диск). Дисковая память на два порядка дешевле ОЗУ в пересчете на бит и зачастую на два порядка больше по величине. У диска есть только одна проблема: случайный доступ к данным на нем занимает примерно на три порядка больше времени. Причиной низкой скорости жесткого диска является тот факт, что диск представляет собой механическую конструкцию, устройство которой продемонстрировано на рис. 1.8.

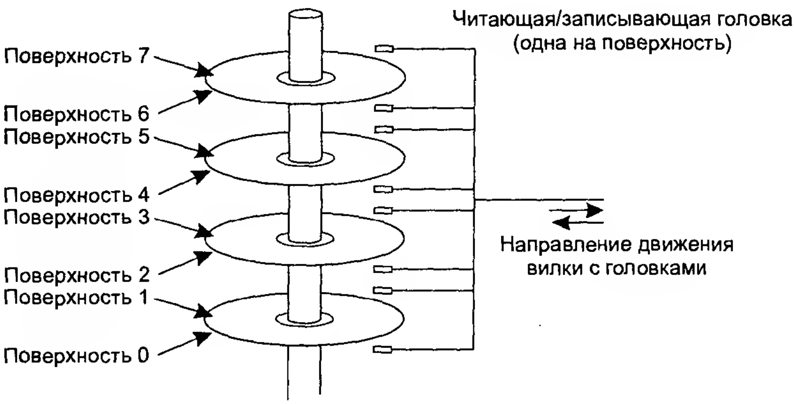


Рис. 1.8. Устройство жесткого диска

Жесткий диск состоит из одной или нескольких металлических пластин, вращающихся со скоростью 5400, 7200 или 10 800 оборотов в минуту. Механическая вилка поворачивается над дисками подобно звукоснимателю на старых граммофонах для проигрывания виниловых пластинок на скорости 33 оборота в минуту. Информация записывается на пластины в виде концентрических окружностей. Головки в каждой заданной позиции вилки могут прочитать кольцо на пластине, называемое **дорожкой**. Все вместе дорожки для заданной позиции вилки формируют **цилиндр**.

Каждая дорожка разделена на некоторое количество секторов, обычно по 512 байт на сектор. На современных дисках внешние цилиндры содержат большее количество секторов, чем внутренние. Перемещение головки от одного цилиндра к другому занимает около 1 мс, а перемещение к произвольному цилиндру требует от 5 до 10 мс, в зависимости от диска. Когда головка располагается над правильной

дорожкой, нужно ждать, пока двигатель повернет диск так, чтобы под головкой встал требуемый сектор. Это занимает дополнительно от 5 до 10 мс, в зависимости от скорости вращения диска. Дальше, когда сектор уже находится под головкой, процесс чтения или записи происходит со скоростью от 5 Мбайт/с для низкоскоростных дисков до 160 Мбайт/с для самых высокоскоростных.

Последний слой в пирамиде памяти занимает магнитная лента. Этот носитель часто используется для создания резервных копий пространства жесткого диска или для хранения очень больших наборов данных. Для доступа к информации на ленте ее сначала нужно поместить в устройство для чтения магнитных лент — это может делать человек или робот (автоматическое управление лентами обычно используется при работе с огромными базами). Затем лента перематывается до запрашиваемого блока с информацией. Весь процесс может длиться минуты. Большой плюс лент заключается в том, что они крайне дешевы и мобильны. Это очень важно для резервных копий, которые нужно содержать отдельно, чтобы они сохранились после стихийных бедствий, например пожаров, наводнений, землетрясений и т. д.

Описанная нами иерархия памяти достаточно типична, но в некоторых вариантах могут присутствовать не все уровни или несколько другие их виды (например, оптический диск). В любом случае при движении по иерархии сверху вниз время произвольного доступа значительно увеличивается от устройства к устройству, вместимость растет эквивалентно времени доступа, а стоимость одного бита информации падает столь же быстрыми темпами. Поэтому вполне вероятно, что такая структура памяти будет популярна еще долгие годы.

Кроме описанных выше видов во многих компьютерах есть небольшое количество постоянной памяти с произвольным доступом — в отличие от оперативной памяти, она не теряет свое содержимое при выключении энергии машины. **ПЗУ** (постоянное запоминающее устройство, ROM, Read Only Memory — память только для чтения) программируется в процессе производства и после этого его содержимое нельзя изменить. Такая память достаточно быстра и дешева. На некоторых компьютерах программы начальной загрузки, используемые при запуске компьютера, находятся в ПЗУ. Кроме того, некоторые карты ввода-вывода содержат ПЗУ для управления низкоуровневыми устройствами.

Электрически стираемое ПЗУ (EEPROM, Electrically Erasable ROM) и **флэш-ОЗУ** (flash RAM) также энергонезависимы, но в отличие от ПЗУ их содержимое можно стереть и переписать. Однако запись данных на них требует намного больше времени, чем запись в оперативную память. Поэтому они используются точно так же, как и ПЗУ. Дополнительное преимущество электрически стираемого ПЗУ и флэшОЗУ состоит в том, что с их помощью теперь можно исправить ошибки, содержащиеся в программах.

Существует еще один вид памяти, называемый CMOS и являющийся энергозависимым. Во многих компьютерах CMOS-память используется для хранения текущих даты и времени. CMOS-память и часовая микросхема, отвечающая за отсчет времени, получают питание от маленького аккумулятора, поэтому компьютер всегда показывает правильное время, даже если он был выключен. CMOS также может содержать конфигурационные параметры, например указание, с какого жесткого диска производить загрузку. CMOS-память используется для этих целей, так как она потребляет настолько мало энергии, что установленный на фабрике

аккумулятор часто работает в течение нескольких лет. Однако если аккумулятор начинает выходить из строя, можно подумать, что компьютер стал страдать болезнью Альцгеймера и забывает то, о чем он помнил годами (например, с какого жесткого диска загружать систему).

Теперь более внимательно рассмотрим основную часть оперативной памяти. Зачастую крайне желательно держать сложные программы в памяти целиком. Если, например, одна программа находится в заблокированном состоянии, ожидая окончания операции чтения данных с диска, то другая программа может в это время использовать центральный процессор, что улучшает показатели эксплуатации процессора. Но при одновременном нахождении в памяти нескольких программ возникает необходимость решения двух следующих проблем:

1. Как защитить программы друг от друга, а ядро системы от всех них?
2. Как управлять перемещением программ в памяти?

Возможны различные решения этих вопросов. Но все они предполагают снабжение процессора специальным оборудованием.

Первая проблема достаточно очевидна, но второй вопрос требует пояснения. В процессе компилирования и компоновки программы компилятор и компоновщик не знают, в какую область физической памяти будет загружена программа после завершения процесса. По этой причине они обычно предполагают, что программа начнется с адреса 0, и помещают туда первую инструкцию. Предположим, что первая инструкция считывает из памяти слово, имеющее адрес 10 000, а вся программа и данные к ней были загружены, начиная с адреса 50 000. Тогда при выполнении первой команды появится сообщение об ошибке, поскольку она будет ссылаться на слово по адресу 10 000 вместо 60 000. Для решения этой проблемы нам нужно или «релоцировать» программу во время загрузки, то есть настроить ее, находя все адреса и изменяя их в соответствии с реальной адресацией (это выполнимо, но дорого), или оперативно изменять адресацию во время работы программы.

Простейшее решение показано на рис. 1.9, *а*. На рисунке видно, что компьютер оборудован двумя специальными регистрами: **базовым** и **предельным**. (Заметим, что в этой книге числа, начинающиеся с 0x, являются шестнадцатеричными в соответствии с правилами орфографии языка C, а числа, начинающиеся с нуля — восьмеричными.) Когда программа начинает работать, в базовый регистр загружается адрес начала исполняемого модуля программы, а предельный регистр говорит о том, сколько занимает исполняемый модуль программы вместе с данными. При выборке команды из памяти аппаратура проверяет счетчик команд, и если он меньше, чем предельный регистр, то добавляет к нему значение базового регистра, а сумму передает памяти. Когда программа хочет прочитать слово данных (например, из адреса 10 000), аппаратура автоматически добавляет к этому адресу содержимое базового регистра (например, 50 000) и передает сумму (60 000) памяти. Базовый регистр дает возможность программе ссылаться на любую часть памяти, следующую за хранящимся в нем адресом. Кроме того, предельный регистр запрещает программе обращение к любой части памяти после программы. Таким образом, с помощью этой схемы решаются обе задачи: защиты и перемещения программ. Стоимость решения равна двум новым регистрам и незначительному увеличению времени, затрачиваемого на операцию (уходящего на проверку предела и суммирование).

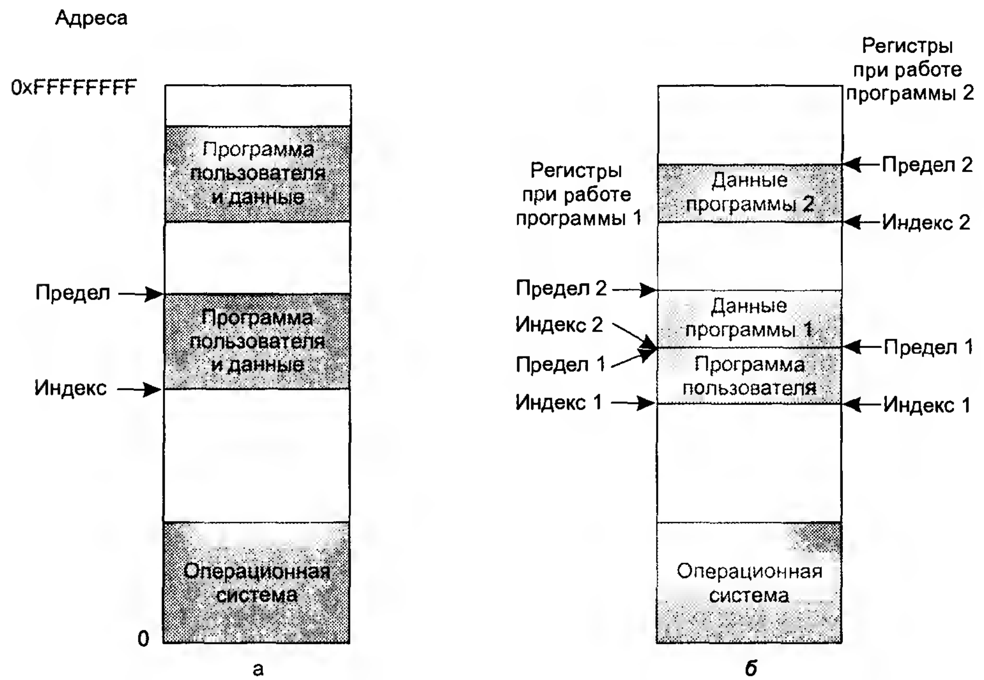


Рис. 1.9. Используется одна пара база—предел. Программа имеет доступ к части памяти, находящейся между базой и пределом (*а*); используются две пары база—предел. Код программы находится между базой 1 и пределом 1, а данные к ней — между базой 2 и пределом 2 (*б*)

В результате проверки и преобразования данных адрес, сформированный программой и называемый **виртуальным**, переводится в адрес, используемый памятью и называемый **физическим**. Устройство, которое выполняет проверку и преобразование, называется устройством управления памятью или **диспетчером памяти** (**MMU**, Memory Management Unit). Диспетчер памяти располагается или в схеме процессора, или близко к ней, но логически находится между процессором и памятью.

Более сложный диспетчер памяти показан на рис. 1.9, *б*. Здесь диспетчер памяти состоит из двух пар базового и предельных регистров: одна пара для текста программы, другая — для данных. Командный регистр и все другие ссылки на текст программы работают с парой 1, а ссылки на данные используют пару 2. Появляется возможность делить одну и ту же программу между несколькими пользователями и при этом хранить в памяти только одну копию программы, что было невозможно в первой схеме. Когда работает программа 1, четыре регистра расположены так, как показано стрелками на рис. 1.9, *б* слева. При работе программы 2 они располагаются так, как показано стрелками на рисунке справа. На самом деле существуют намного более сложные диспетчеры памяти, мы изучим их позже в этой книге. А сейчас нужно запомнить, что управление диспетчером памяти должно быть функцией операционной системы, так как нет уверенности, что пользователь сделает это корректно.

На характеристики памяти в основном влияют два аспекта. Во-первых, кэш скрывает относительно низкую скорость памяти. После того как программа про-

работала некоторое время, кэш заполняется строками программы, увеличивая скорость. Однако когда операционная система переключается от одной программы к другой, кэш остается заполненным данными первой программы, а необходимые строки новой программы должны загружаться уже из физической памяти. Такая операция может стать главной причиной снижения производительности, если она происходит слишком часто.

Во-вторых, при переключении от одной программы к другой регистры управления памятью должны меняться. На рис. 1.9, *б* требуется перезагрузка только четырех регистров, что не является серьезной проблемой, но в реальных диспетчерах памяти должно перезагружаться, явно или динамически, намного большее количество регистров. В любом случае подобная операция занимает некоторое время. Мораль этой истории такова: переключение от одной программы к другой, называемое **переключением контекста**, очень дорого.

Устройства ввода-вывода

Память не является единственным ресурсом, которым должна управлять операционная система. Устройства ввода-вывода также тесно взаимодействуют с операционной системой. Как видно из рис. 1.5, устройства ввода-вывода обычно состоят из двух частей: контроллера и самого устройства. Контроллер — это микросхема или набор микросхем на вставляемой в разъем плате, физически управляющая устройством. Он принимает команды операционной системы, например указание прочитать данные с устройства, и выполняет их.

Во многих случаях фактическое управление устройством очень сложно и требует высокого уровня детализации, поэтому в работу контроллера входит представление простого интерфейса для операционной системы. Контроллер диска может принять команду прочесть сектор 11 206 с диска 2. При этом контроллер должен преобразовать линейный номер сектора в номер цилиндра, сектора и головки. Операция преобразования усложняется тем фактом, что внешние цилиндры могут иметь больше секторов, чем внутренние, и что номера испорченных секторов отображаются на другие секторы. Затем контроллер должен определить, над каким цилиндром находится в данный момент головка, и дать ей последовательность импульсов, чтобы переместить ее на необходимое количество цилиндров. Дальше нужно ждать, пока повернется диск, поместив требуемый сектор под головку. Затем начинается чтение и сохранение битов по мере поступления их с диска, удаление заголовка и вычисление контрольной суммы. Наконец, контроллер должен собрать полученные биты в слова и сохранить их в памяти. Для осуществления всей этой работы контроллеры часто содержат маленькие встроенные компьютеры, запрограммированные на выполнение подобных задач.

Следующей частью является само устройство. Устройства имеют достаточно простые интерфейсы, во-первых, потому что их возможности весьма невелики и, во-вторых, потому что нужно привести их к единому стандарту. Единый стандарт необходим, чтобы любой IDE-контроллер диска мог управлять любым IDE-диском. Аббревиатура **IDE** образована от **Integrated Drive Electronics** (встроенный интерфейс накопителей). IDE-интерфейс является стандартным для дисков на компьютерах с процессором Pentium, а также некоторых других компьютерах.

Поскольку настоящий интерфейс устройства скрыт с помощью контроллера, операционная система видит только интерфейс контроллера, который может сильно отличаться от интерфейса самого устройства.

Так как все типы контроллеров отличаются друг от друга, для управления ими требуется различное программное обеспечение. Программа, которая общается с контроллером, отдает ему команды и получает ответы, называется **драйвером устройства**. Каждый производитель контроллеров должен поставлять драйверы для поддерживаемых им операционных систем. Вы можете купить сканер с драйверами для Windows 98, Windows 2000 и UNIX. Если вы хотите получить возможность использовать драйвер, его нужно установить в операционную систему так, чтобы он мог работать в режиме ядра. Теоретически драйверы могут работать вне ядра, но такую возможность поддерживают всего несколько существующих систем, так как для этого требуется, чтобы драйвер в пространстве пользователя имел доступ к устройству неким контролируемым способом — очень редко поддерживаемое свойство. Есть три способа установки драйвера в ядро. Первый заключается в том, чтобы заново скомпоновать ядро вместе с новым драйвером и затем перезагрузить систему. Так работает множество систем UNIX. Второй: создать запись во входящем в операционную систему файле, говорящую о том, что требуется драйвер, и затем перезагрузить систему. Во время начальной загрузки операционная система сама находит нужные драйверы и загружает их. Так работает система Windows. При третьем способе операционная система может принимать новые драйверы, не прерывая работы, и оперативно устанавливать их, не нуждаясь при этом в перезагрузке. Этот способ редко используется, но сейчас он становится все более и более распространенным. Такие съемные устройства, как шины USB и IEEE 1394 (мы поговорим о них ниже), всегда нуждаются в динамически загружаемых драйверах.

Для связи с каждым контроллером существует небольшое количество регистров. Например, минимальный контроллер диска может иметь регистры для определения адреса на диске, адреса в памяти, номер сектора и направления операции (чтение или запись). Чтобы активизировать контроллер, драйвер получает команду от операционной системы, затем транслирует ее в величины, подходящие для записи в регистры устройства.

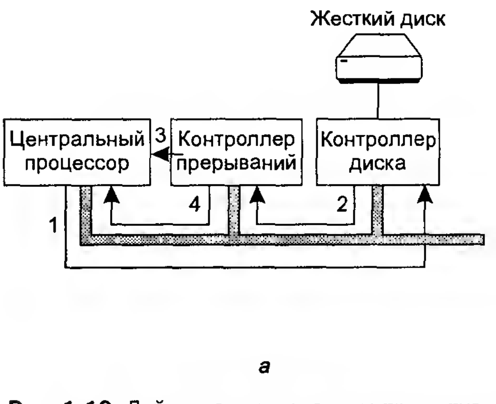
На некоторых компьютерах регистры устройств отображаются в адресное пространство операционной системы, поэтому их можно читать или записывать как обычные слова в памяти. На таких машинах не нужны специальные команды ввода-вывода, а программы пользователей можно оградить от аппаратуры, помещая эти адреса в памяти за пределами досягаемости программ (например, с помощью базового и предельного регистров). На других компьютерах регистры устройств располагаются в специальных портах ввода-вывода, и каждый регистр имеет свой адрес порта. На этих машинах в режиме работы ядра доступны команды `IN` и `OUT`, они позволяют драйверам считывать и записывать регистры. Первая схема устраняет необходимость специальных команд ввода-вывода, но использует некоторое количество адресного пространства. Вторая схема не затрагивает адресное пространство, но требует наличие специальных команд. Обе схемы широко используются.

Ввод и вывод данных можно осуществлять тремя различными способами. Простейший метод состоит в том, что пользовательская программа выдает системный запрос, который ядро транслирует в вызов процедуры соответствующего драйве-

ра. Затем драйвер начинает процесс ввода-вывода. В это время драйвер выполняет очень короткий программный цикл, постоянно опрашивая готовность устройства, с которым он работает (обычно есть некий бит, который указывает на то, что устройство все еще занято). По завершении операции ввода-вывода драйвер помещает данные туда, куда требуется, и возвращается в исходное состояние. Затем операционная система возвращает управление программе, осуществлявшей вызов. Этот метод называется **ожиданием готовности** или **активным ожиданием** и имеет один недостаток: процессор должен опрашивать устройство до тех пор, пока оно не завершит свою работу.

При втором способе драйвер запускает устройство и просит его выдать прерывание по окончании ввода-вывода. После этого драйвер возвращает данные, операционная система блокирует программу вызова, если это нужно, и начинает выполнять другие задания. Когда контроллер обнаруживает окончание передачи данных, он генерирует **прерывание**, чтобы сигнализировать о завершении операции.

Прерывания очень важны в работе операционной системы, поэтому рассмотрим это понятие более внимательно. На рис. 1.10, *а* показан трехшаговый процесс ввода-вывода. На первом шаге драйвер передает команду контроллеру, записывая информацию в регистры устройства. Затем контроллер запускает устройство. Когда контроллер заканчивает чтение или запись того количества байтов, которое ему было указано передать, он посылает сигнал микросхеме контроллера прерываний, используя определенные провода шины, — это шаг *2*. На шаге *3*, если контроллер прерываний готов к приему прерывания (а этого может и не быть, если он занят прерыванием более высокого приоритета), то он подает сигнал на определенный контакт процессора, таким образом информируя центральный процессор. На шаге *4* контроллер прерываний выставляет номер устройства на шину так, чтобы центральный процессор мог прочесть его и узнать, какое устройство только что завершило свою работу (ведь в одно и то же время могут работать несколько устройств).



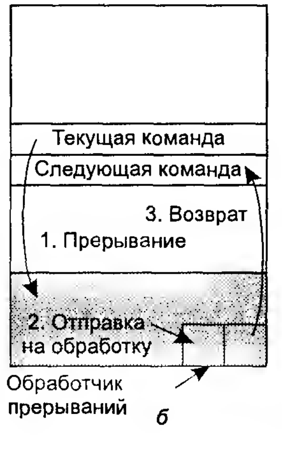


Рис. 1.10. Действия, выполняемые при запуске устройства ввода-вывода и получении прерывания (*а*); обработка прерывания включает в себя получение прерывания, переход к обработчику прерываний и возврат к программе пользователя (*б*)

Как только центральный процессор решил принять прерывание, содержимое счетчика команд (PC) и слова состояния процессора (PSW) помещается в текущий стек, а процессор переключается в режим работы ядра. Номер устройства может использоваться как индекс части памяти, служащий для поиска адреса обработчика прерываний данного устройства. Эта часть памяти называется **вектором прерываний**. Когда обработчик прерываний (это часть драйвера устройства, пославшего прерывание) начинает свою работу, он удаляет расположенные в стеке счетчик команд и слово состояния процессора, сохраняет их и запрашивает устройство, чтобы получить информацию о его состоянии. После того как обработка прерывания целиком завершена, управление возвращается к работавшей до этого программе пользователя, к той команде, выполнение которой еще не было закончено. Описанные шаги показаны на рис. 1.10, *б*.

Третий метод ввода-вывода информации заключается в использовании специального контроллера **прямого доступа к памяти (DMA**, Direct Memory Access), который управляет потоком битов между оперативной памятью и некоторыми контроллерами без постоянного вмешательства центрального процессора. Процессор вызывает микросхему DMA, говорит ей, сколько байтов нужно передать, сообщает адреса устройства и памяти, а также направление передачи данных и позволяет дальше действовать ей самой. По завершении работы DMA инициирует прерывание, которое обрабатывается так же, как было описано выше. Контроллер DMA и аппаратура ввода-вывода более детально будут обсуждаться в главе 5.

Прерывания часто происходят в очень неподходящие моменты, например во время обработки другого прерывания. По этой причине центральный процессор обладает возможностью запрещать прерывания и разрешать их позже. Пока прерывания запрещены, все устройства, завершившие работу, продолжают посылать свои сигналы, но работа процессора не прерывается до тех пор, пока прерывания не будут разрешены. Если заканчивают работу сразу несколько устройств в то время, когда прерывания запрещены, контроллер прерываний решает, какое из них должно быть обработано первым, обычно основываясь на статических приоритетах, назначенных для каждого устройства. Устройство с высшим приоритетом побеждает.

Шины

Структура, показанная на рис. 1.5, в течение многих лет использовалась на миникомпьютерах, а также на первых моделях IBM PC. Но поскольку процессоры и память стали работать быстрее, возможности одной шины (и, конечно, шины IBM PC) по управлению всей передачей данных достигли своего предела. Нужно было что-то делать. В результате в систему добавились дополнительные шины как для ускорения общения с устройствами ввода-вывода, так и для пересылки данных между процессором и памятью. Вследствие этой эволюции сейчас большая система Pentium выглядит примерно так, как изображено на рис. 1.11.

У этой системы восемь шин (шина кэша, локальная шина, шина памяти, PCI, SCSI, USB, IDE и ISA), каждая со своей скоростью передачи данных и своими функциями. В операционной системе для управления компьютером и его конфигурации должны находиться сведения обо всех этих шинах. Две основные шины — это **ISA** (Industry Standard Architecture — промышленная стандартная архитекту-

ра), оригинальная шина компьютера IBM PC, и ее преемник, шина **PCI** (Peripheral Component Interconnect — интерфейс периферийных устройств). Шина ISA впервые появилась на компьютерах IBM PC/AT, она работает на частоте 8,33 МГц и может передавать два байта за такт с максимальной скоростью 16,67 Мбайт/с. Она включена в систему для обратной совместимости со старыми медленными платами ввода-вывода. Шина PCI была создана компанией Intel в качестве преемницы шины ISA. Она может работать на частоте 66 МГц и передавать сразу по 8 байт за такт со скоростью 528 Мбайт/с. Сейчас большинство высокоскоростных устройств ввода-вывода используют шины PCI. Даже некоторые компьютеры с процессорами, отличными от Intel, пользуются шиной PCI, поскольку с ней совместимо очень много плат ввода-вывода.

При такой конфигурации центральный процессор по локальной шине передает данные микросхеме PCI-моста, который, в свою очередь, обращается к памяти по выделенной шине памяти, часто работающей на частоте 100 МГц. Системы Pentium имеют кэш первого уровня (кэш L1), встроенный в процессор, и намного больший внешний кэш второго уровня (кэш L2), подключенный к процессору отдельной шиной кэша.

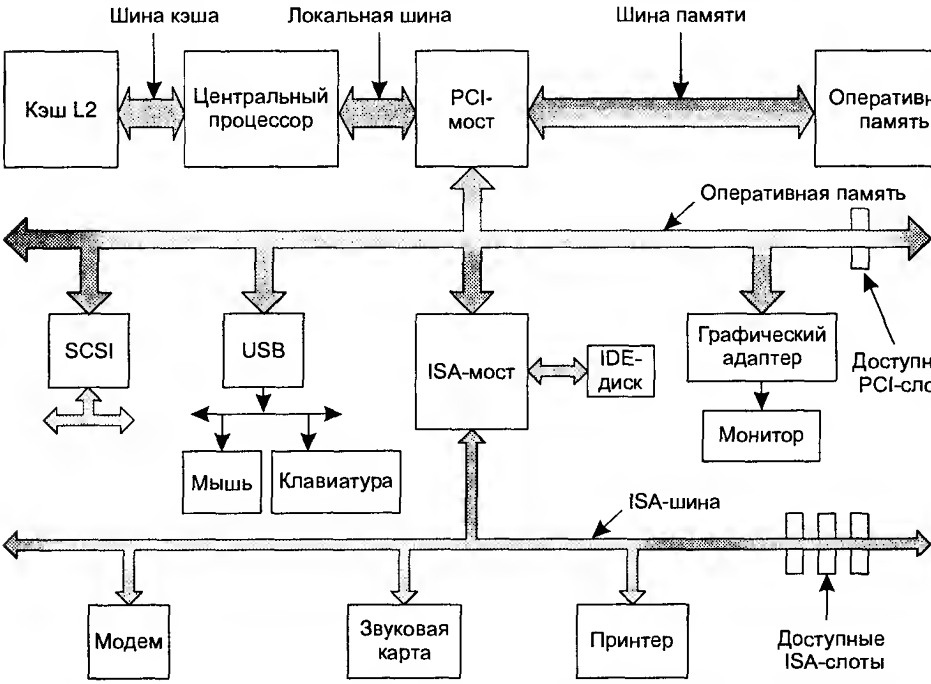


Рис. 1.11. Структура большой системы Pentium

Кроме того, в систему входят три специализированные шины: IDE, USB и SCSI. Шина IDE служит для присоединения периферийных устройств к системе — дисков и устройств для чтения компакт-дисков (CD-ROM). IDE-шина — это потомок интерфейса контроллера диска на PC/AT, и сейчас она входит в стандартный комплект всех систем, основанных на процессорах Pentium.

Шина **USB** (Universal Serial Bus — универсальная последовательная шина) была придумана для того, чтобы присоединить к компьютеру все медленные устройства ввода-вывода, такие как клавиатура и мышь. Она использует маленький четырехпроводной разъем, причем два провода поставляют электропитание к USB-устройствам. USB — это централизованная шина, по которой главное устройство каждую миллисекунду опрашивает устройства ввода-вывода, чтобы узнать, есть ли у них данные. Она может управлять загрузкой данных со скоростью 1,5 Мбайт/с. Все USB-устройства используют один драйвер, избавляя нас тем самым от необходимости установки новых драйверов для каждого нового USB-устройства. Поэтому USB-устройства можно присоединять к системе без ее перезагрузки.

SCSI (Small Computer System Interface — системный интерфейс малых компьютеров) — это высокопроизводительная шина, применяемая для быстрых дисков, сканеров и других устройств, нуждающихся в значительной пропускной способности. Ее производительность достигает 160 Мбайт/с. Шина SCSI используется в системах Macintosh с момента их появления, кроме того, она популярна в UNIX-системах и некоторых системах на базе процессоров Intel.

Есть еще одна шина (не показанная на рис. 1.11), называется **IEEE 1394**. Иногда ее также называют FireWire, хотя, строго говоря, FireWire — это название, данное компанией Apple собственной реализации шины 1394. Как и USB, IEEE 1394 является бит-последовательной шиной, но она поддерживает пакетную передачу данных со скоростью, достигающей 50 Мбайт/с. Это ее свойство позволяет подключать к компьютеру портативные цифровые видеокамеры и тому подобные мультимедийные устройства. В отличие от USB шина IEEE 1394 не имеет центрального контроллера. Шины SCSI и IEEE 1394 конкурируют с разработанной более быстрой версией шины USB.

Работая в окружении, изображенном на рис. 1.11, операционная система должна уметь распознавать аппаратные составляющие и уметь их настраивать. Это требование привело компании Intel и Microsoft к разработке системы персонального компьютера, называемой **plug and play** («включи и работай»). В основе этой системы лежала концепция, близкая к той, что была впервые реализована компанией Apple Macintosh. До появления plug and play каждая плата ввода-вывода имела фиксированные адреса регистров ввода-вывода и уровень запроса прерывания. Например, клавиатура использовала прерывание 1 и адреса в диапазоне от 0x60 до 0x64; контроллер гибкого диска использовал прерывание 6 и адреса от 0x3F0 до 0x3F7; принтер пользовался прерыванием 7 и адресами от 0x378 до 0x37A и т. д.

Все в этой схеме было хорошо до тех пор, пока пользователь не покупал звуковую карту и модем, и оказывалось, что оба устройства случайно использовали, скажем, прерывание 4. В таком случае они конфликтовали и не могли работать вместе. Возможным решением было встроить набор DIP-переключателей или джамперов (jumper — перемычка) в каждую плату и объяснить пользователю необходимость настройки каждой платы таким образом, чтобы адреса портов и номера прерываний различных устройств не конфликтовали друг с другом. Подростки, посвятившие свою жизнь изучению тонкостей аппаратуры персонального компьютера, иногда могут сделать это без ошибок. К сожалению, кроме них это практически никому не удавалось, что приводило к полному хаосу.

Стандарт plug and play позволяет системе автоматически собирать информацию об устройствах ввода-вывода, централизованно назначать уровни прерывания и адреса ввода-вывода, а затем сообщать каждой плате эту информацию — вот краткое описание процесса. Такая система работает на компьютерах Pentium. Каждый компьютер с процессором Pentium содержит материнскую плату (в США благодаря успехам борьбы за политическую корректность эту плату теперь решено называть родительской). На материнской плате находится программа, называемая системой **BIOS** (Basic Input Output System — базовая система ввода-вывода). BIOS содержит программы ввода-вывода низкого уровня, включая процедуры для чтения с клавиатуры, вывода информации на экран, ввода-вывода данных с диска и т. д. В настоящее время эти функции хранятся во флэш-ОЗУ, которая в обычных условиях является неизменяемой, но, если в BIOS нашлись какие-либо ошибки, ее может изменить операционная система.

При начальной загрузке компьютера стартует система BIOS. Сначала она проверяет количество установленной в системе оперативной памяти, подключены ли клавиатура и другие основные устройства и корректно ли они отзываются. BIOS начинает проверку с шин ISA и PCI, чтобы определить все устройства, присоединенные к ним. Некоторые из этих устройств являются традиционными, их также называют **унаследованными** (legacy), то есть созданными до изобретения plug and play. Они имеют фиксированные уровни прерывания и адрес порта ввода-вывода (например, заданные с помощью переключателей или перемычек на плате ввода-вывода без возможности их изменения операционной системой). Эти устройства регистрируются. Устройства plug and play тоже регистрируются. Если присутствующие устройства отличаются от тех, что были во время последней загрузки, конфигурируются новые устройства.

Затем BIOS определяет устройство, с которого будет происходить загрузка, по очереди пробуя каждое из списка, хранящегося в CMOS-памяти. Пользователь может изменить этот список, войдя в конфигурационную программу BIOS сразу после загрузки. Обычно сначала делается попытка загрузиться с гибкого диска. Если это не удается, пробуется компакт-диск. Если в компьютере отсутствуют и гибкий диск, и компакт-диск, система загружается с жесткого диска. С загрузочного устройства считывается в память и выполняется первый сектор. В этом секторе находится программа, обычно проверяющая таблицу разделов в конце загрузочного сектора, чтобы определить, который из разделов является активным. Затем из того же раздела читается вторичный загрузчик. Он считывает из активного раздела операционную систему и запускает ее.

После этого операционная система опрашивает BIOS, чтобы получить информацию о конфигурации компьютера. Для каждого устройства она проверяет наличие драйвера. Если драйвер отсутствует, операционная система просит пользователя вставить гибкий диск или компакт-диск, содержащий драйвер (эти диски поставляются производителем устройства). Если же все драйверы на месте, операционная система загружает их в ядро. Затем она инициализирует таблицы драйверов, создает все необходимые фоновые процессы и запускает программу ввода пароля или графический интерфейс на каждом терминале. По крайней мере, предполагается, что операционная система должна работать таким образом. В реальной жизни система plug and play часто бывает настолько ненадежна, что многие люди называют ее plug and pray («включи и молись»).

Понятия операционной системы

Для каждой операционной системы существует набор базовых понятий, например процессы, память и файлы, которые являются самыми важными для понимания общей идеи. В следующих разделах мы рассмотрим некоторые из них по возможности кратко, как это должно быть сделано во введении. Позже мы вернемся к каждому из них и рассмотрим в деталях. Для иллюстрации этих понятий время от времени мы будем использовать примеры, в основном взятые из системы UNIX. Но аналогичные примеры можно найти и в других системах.

Процессы

Ключевое понятие операционной системы — **процесс**. Процессом, по существу, называют программу в момент выполнения. С каждым процессом связывается его **адресное пространство** — список адресов в памяти от некоторого минимума (обычно нуля) до некоторого максимума, которые процесс может прочесть и в которые он может писать. Адресное пространство содержит саму программу, данные к ней и ее стек. Со всяким процессом связывается некий набор регистров, включая счетчик команд, указатель стека и другие аппаратные регистры, плюс вся остальная информация, необходимая для запуска программы.

Мы более детально рассмотрим понятие процесса в главе 2, но сейчас для того, чтобы интуитивно осознать, что это такое, вспомним о системах, работающих в режиме разделения времени. Предположим, периодически операционная система решает остановить работу одного процесса и запустить другой, потому что первый израсходовал отведенную для него часть рабочего времени центрального процессора в прошедшую секунду.

Если процесс был приостановлен подобным образом, позже он должен быть запущен заново из того же состояния, в каком его остановили. Следовательно, всю информацию о процессе нужно где-либо явно сохранить на время его приостановки. Например, процесс может иметь открытыми для чтения несколько файлов одновременно. Связанный с каждым файлом указатель дает текущую позицию (то есть номер байта или записи, которые будут прочитаны следующими). При временном прекращении процесса все указатели нужно сохранить так, чтобы команда чтения, выполненная после возобновления процесса, прочла правильные данные. Во многих операционных системах вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в таблице операционной системы. Эта таблица называется **таблицей процессов** и представляет собой массив (или связанный список) структур, по одной на каждый существующий в данный момент процесс.

Таким образом, приостановленный процесс состоит из собственного адресного пространства, обычно называемого **образом памяти** (core image, core в переводе означает «сердечник», в честь использовавшейся давным-давно памяти на магнитных сердечниках), и компонентов таблицы процесса, содержащей, помимо других величин, его регистры.

Главными системными вызовами, управляющими процессами, являются вызовы, связанные с созданием и окончанием процессов. Рассмотрим типичный пример.

Процесс, называемый **интерпретатором команд** или **оболочкой** (shell), читает команды с терминала. Пользователь только что напечатал команду, содержащую запрос на компиляцию программы. Теперь оболочка должна создать новый процесс, который запустит компилятор. Когда процесс закончит компиляцию, он выполнит системный вызов, завершающий его собственную работу.

Если процесс может создавать несколько других процессов (называющихся **дочерними процессами**), а эти процессы, в свою очередь, тоже могут создать дочерние процессы, перед нами предстает дерево процессов, изображенное на рис. 1.12. Связанные процессы — это те, которые объединены для выполнения некоторой задачи, и им нужно часто передавать данные от одного к другому и синхронизировать свою деятельность. Такая связь называется **межпроцессным взаимодействием** и будет обсуждена в деталях в главе 2.

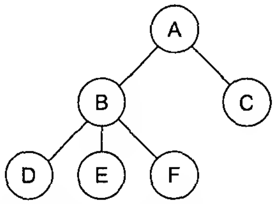


Рис. 1.12. Дерево процесса. Процесс A создал два дочерних процесса B и C. Процесс B создал три дочерних процесса D, E и F

Другие системные вызовы предназначаются для запросов о предоставлении дополнительной памяти (или освобождении не использующейся памяти), ожидании завершения дочерних процессов и наложении одной программы на другую.

Время от времени необходимо передавать информацию работающему процессу так, чтобы он не простаивал в ожидании получения этой информации. Например, процесс, связанный с другим процессом на удаленном компьютере, делает это, посылая сообщения по сети. Чтобы предотвратить возможность потери сообщения или ответа на него, отправитель может потребовать от собственной операционной системы уведомления, если по истечении определенного интервала ожидания не будет получено подтверждение о получении сообщения. В этом случае он сможет повторить отправку сообщения. После установки таймера программа продолжит выполнение другой работы.

Если по истечении определенного количества секунд ответа нет, операционная система посылает процессу **сигнал тревоги**. Сигнал вызывает временную остановку работы процесса независимо от того, что процесс делает в данный момент; сохраняет его регистры в стеке и запускает специальную процедуру обработки сигнала (например, передающую повторно предположительно потерянное сообщение). После завершения обработки сигнала работающий процесс запускается заново в том состоянии, в котором он находился до сигнала. Сигналы являются программными аналогами аппаратных прерываний и могут быть сгенерированы по различным причинам, а не только из-за истечения какого-либо интервала времени. Многие аппаратные прерывания (например, вызванные выполнением недопустимой команды или использованием неправильного адреса) также преобразуются в сигналы процессу, в котором произошла ошибка.

Каждому пользователю, которому разрешено пользоваться системой, системный администратор присваивает **UID** (User IDentification — идентификатор пользователя). У каждого работающего процесса есть идентификатор пользователя, запустившего его. Дочерний процесс получает тот же самый UID, что и его родитель. Пользователи могут становиться членами групп, каждая из которых имеет **идентификатор группы** (GID, Group IDentification).

Пользователь с особым идентификатором UID, называемый в UNIX **«суперпользователем»** (superuser), имеет особые полномочия и может игнорировать множество правил защиты. В огромных системах только системный администратор знает пароль, необходимый для того, чтобы стать суперпользователем. Однако множество обыкновенных пользователей (особенно студенты) тратят значительное количество времени и труда на то, чтобы найти брешь в системе, которая позволит им стать суперпользователями без пароля.

Мы изучим процессы, взаимодействие между ними и связанные с ними вопросы в главе 2.

Взаимоблокировка

Когда взаимодействуют два или более процессов, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи. Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.

Тупиковую ситуацию легче всего представить с помощью примера из реального мира, с которым знаком каждый, — это пробки на дорогах. Обсудим ситуацию на рис. 1.13, *а*. Четыре автобуса приближаются к перекрестку. За каждым автобусом есть еще машины (они не показаны на рисунке). При определенном невезении первые четыре автобуса прибудут на перекресток одновременно, что приведет к ситуации, показанной на рис. 1.13, *б*. Все автобусы заблокировали друг друга, поскольку ни один автобус не может двигаться вперед. Каждый автобус блокирует остальные. При этом они не могут двигаться назад, потому что за ними есть еще автобусы. И нет простого способа выпутаться из этой ситуации.

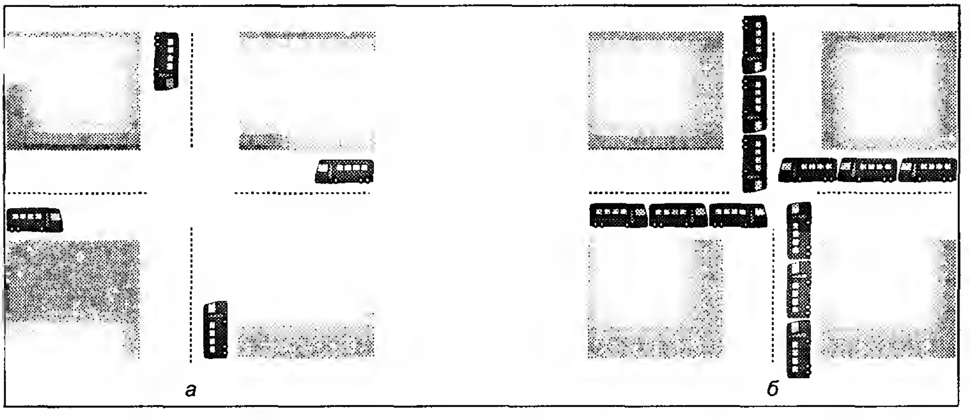


Рис. 1.13. Потенциальная взаимоблокировка (*а*); фактическая взаимоблокировка (*б*)

Компьютерные процессы могут попадать в аналогичные ситуации, в которых они не могут продвигаться дальше. Представьте себе компьютер с накопителем на магнитной ленте и записывающим компакт-диски устройством (CD-recorder). Теперь представьте, что каждому из двух процессов нужно записать данные с ленты на компакт-диск. Процесс 1 запрашивает и получает в пользование устройство с лентой. Затем процесс 2 запрашивает и получает устройство для записи компакт-дисков. После этого процесс 1 запрашивает устройство для записи компакт-дисков и приостанавливается до тех пор, пока процесс 2 не освободит его. Наконец, процесс 2 запрашивает устройство с лентой и также останавливается на время, потому что магнитофон уже занят процессом 1. Перед нами типичный тупик, из которого нет выхода. Мы в деталях изучим тупиковые ситуации и посмотрим, что можно с ними делать в главе 3.

Управление памятью

В каждом компьютере есть оперативная память, используемая для хранения выполняющихся программ. В очень простых операционных системах в конкретный момент времени в памяти может находиться только одна программа. Для запуска второй программы сначала нужно удалить из памяти первую и загрузить на ее место вторую.

Более изощренные системы позволяют одновременно находиться в памяти нескольким программам. Для того чтобы они не мешали друг другу (и операционной системе), необходим некий защитный механизм. Хотя этот механизм располагается в аппаратуре, он управляется операционной системой.

Вышеизложенная точка зрения имеет отношение к управлению оперативной памятью компьютера и к ее защите. Другой, но не менее важный, связанный с памятью вопрос — это управление адресным пространством процессов. Обычно под каждый процесс отводится некоторый набор адресов, которые он может использовать, чаще всего начинающийся с 0 и продолжающийся до некого максимума. В простейшем случае максимальная величина адресного пространства для процесса меньше основной памяти. Тогда процесс может заполнить свое адресное пространство, и памяти хватит на то, чтобы содержать его целиком.

Однако на многих компьютерах адресация 32- или 64-разрядная, что дает для пространства адресов 2^{32} и 2^{64} байтов соответственно. Что произойдет, если адресное пространство процесса окажется больше, чем оперативная память компьютера, и процесс захочет использовать его целиком? На первых компьютерах подобным процессам просто не везло. В наши дни существует метод, называемый виртуальной памятью, при котором операционная система держит часть адресов в оперативной памяти, а часть на диске и меняет их местами при необходимости. Эта важная функция операционной системы, а также другие понятия, связанные с управлением памятью, будут рассмотрены в главе 4.

Ввод-вывод данных

Во всех компьютерах есть физическое устройство для получения входных данных и вывода информации. Посудите сами, что хорошего было бы в компьютере, если бы пользователи не могли сказать ему, что делать, и не могли получить результа-

ты после завершения выполненной работы? Существует много видов устройств ввода-вывода, например клавиатуры, мониторы, принтеры и т. д. Всеми ими должна управлять операционная система.

Каждая операционная система имеет свою подсистему ввода-вывода для управления устройствами ввода-вывода. Некоторые из программ ввода-вывода являются независимыми от устройств, то есть их можно применить ко многим или ко всем устройствам ввода-вывода. Другая часть программного обеспечения ввода-вывода, в которую входят драйверы устройств, предназначена для определенных устройств ввода-вывода. В главе 5 мы рассмотрим программное обеспечение ввода-вывода данных.

Файлы

Файловая система — это еще одно ключевое понятие, поддерживаемое виртуально всеми операционными системами. Как было замечено ранее, основной функцией операционной системы является скрытие особенностей дисков и других устройств ввода-вывода и предоставление пользователю понятной и удобной абстрактной модели независимых от устройств файлов. Системные вызовы очевидно необходимы для создания, удаления, чтения или записи файлов. Перед тем как прочитать файл, его нужно разместить на диске и открыть, а после прочтения его нужно закрыть. Все эти функции осуществляют системные вызовы.

Предоставляя место для хранения файлов, операционные системы используют понятие **каталога (directory)** как способ объединения файлов в группы. Например, студент может иметь по одному каталогу для каждого изучаемого им курса (для программ, необходимых в рамках этого курса), каталог для электронной почты, и еще один — для своей домашней web-страницы. Для создания и удаления каталогов также необходимы системные вызовы. Они же обеспечивают перемещение существующего файла в каталог и удаление файла из каталога. Содержимое каталогов могут составлять файлы или другие каталоги. Эта модель создает структуру — файловую систему, — как показано на рис. 1.14.

Иерархии процессов и файлов организованы в виде деревьев, но на этом сходство заканчивается. Иерархия процессов обычно не очень глубока (в ней редко бывает больше трех уровней), тогда как файловая структура достаточно часто имеет четыре, пять или даже больше уровней в глубину. Иерархия процессов обычно живет очень недолго, как правило, несколько минут, иерархия каталогов может существовать годами. Принадлежность и защита также различны для процессов и файлов. Обычно только родительский процесс может управлять или даже просто иметь доступ к дочернему процессу, однако практически всегда существует механизм, позволяющий читать файлы и каталоги не только владельцу файла, а более широкой группе пользователей.

Каждый файл в иерархии каталогов можно определить, задав его **имя пути**, называемое также полным именем файла. Путь начинается из вершины структуры каталогов, называемой **корневым каталогом**. Такое абсолютное имя пути состоит из списка каталогов, которые нужно пройти от корневого каталога к файлу, с разделением отдельных компонентов косой чертой. На рис. 1.14 путь к файлу CS101 выглядит как */Faculty/Prof.Brown/Courses/CS101*. Первая косая черта гово-

рит о том, что этот путь — абсолютный, то есть начинается от корневого каталога. В MS-DOS и Windows для разделения компонентов вместо символа косой черты используется обратная косая черта (\). Тогда этот путь будет выглядеть так: *\Faculty\Prof.Brown\Courses\CS101*. В нашей книге для записи пути мы в основном будем использовать соглашения UNIX.

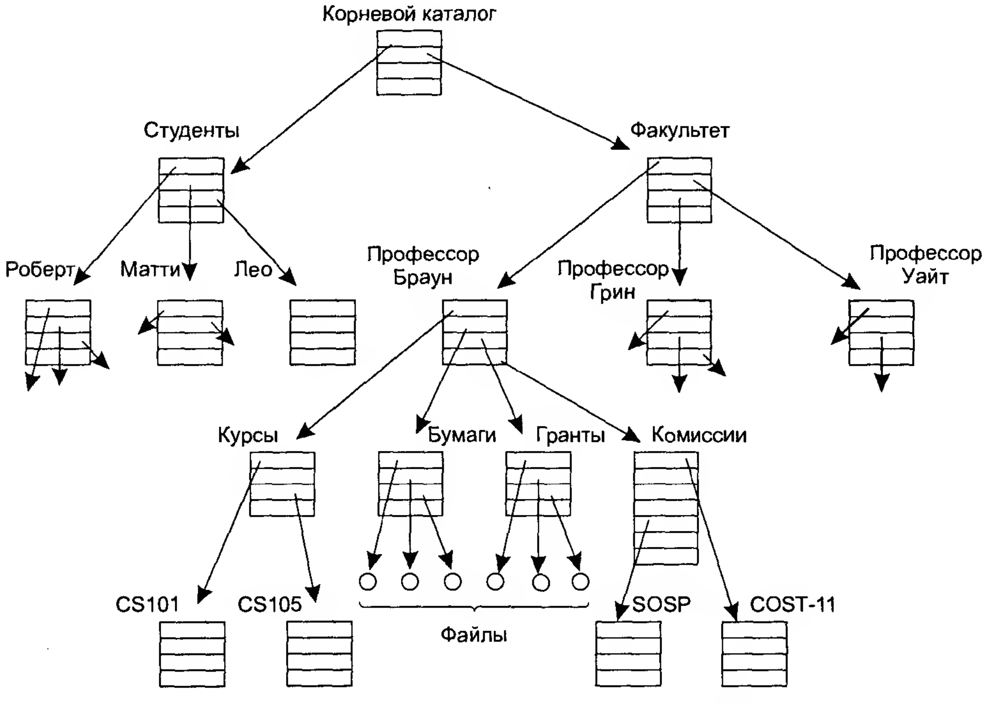


Рис. 1.14. Файловая система факультета университета

В каждый момент времени у каждого процесса есть текущий **рабочий каталог**, в котором ищутся пути файлов, не начинающиеся с косой черты. Например, если на рис. 1.14 */Faculty/Prof.Brown* является рабочим каталогом, то использование пути *Courses/CS101* даст тот же самый файл, что и абсолютный путь, написанный выше. Процессы могут изменять свой рабочий каталог, используя системные вызовы.

Перед тем как прочесть или записать файл, его нужно открыть, в это же время проверяется разрешение доступа. Если доступ разрешен, система возвращает небольшое целое число, называемое **дескриптором файла** и используемое в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Другое важное понятие в UNIX — это установленная (смонтированная) файловая система. Почти все персональные компьютеры имеют один или два дисковода для гибких дисков, куда можно вставить и откуда можно вынуть диск. Чтобы предоставить возможность общения со сменными носителями (включая компакт-диски), UNIX позволяет присоединять файловую систему сменного диска к главному дереву. Рассмотрим ситуацию на рис. 1.15, *а*. Перед вызовом системной процедуры mount **корневая файловая система** на жестком диске и вторая файловая система на гибком диске существуют раздельно и никак не связаны между собой.

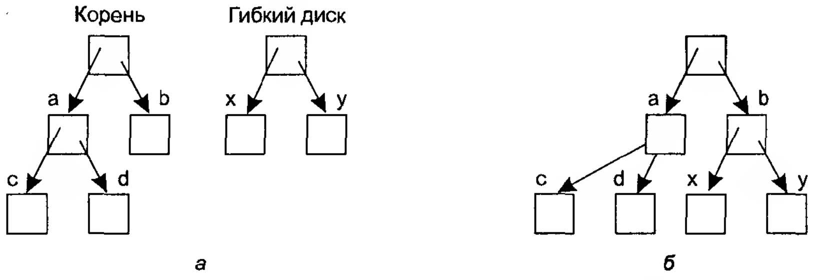


Рис. 1.15. Перед установкой файлы на диске 0 недоступны (*а*); после монтирования они становятся частью общей файловой структуры (*б*)

Однако файлы на гибком диске нельзя использовать, потому что для них невозможно определить путь. UNIX не позволяет присоединять к началу пути название диска или его номер, так как это привело бы к жесткой зависимости от устройств, которой операционная система должна избегать. Вместо этого системный вызов mount позволяет присоединять файловую систему на гибком диске к корневой файловой системе в том месте, где этого захочет программа. На рис. 1.15, *б* файловая система гибкого диска была установлена в каталог *b*, таким образом, обеспечен доступ к файлам по путям */b/x/* и */b/y*. Если каталог *b* содержал какие-либо файлы, они будут недоступны, пока смонтирован гибкий диск, так как теперь */b* ссылается на корневой каталог гибкого диска. (Невозможность доступа к этим файлам не так страшна, как кажется с первого взгляда: файловые системы почти всегда устанавливаются в пустые каталоги.) Если система содержит несколько жестких дисков, они все могут быть встроены в одно дерево таким же образом.

Еще одно важное понятие в UNIX — это **специальный файл**. Специальные файлы служат для того, чтобы устройства ввода-вывода выглядели как файлы. При этом можно прочесть информацию из специальных файлов или записать ее туда с помощью тех же самых системных вызовов, что используются для чтения и записи файлов. Существует два вида специальных файлов: **блочные специальные файлы** и **символьные специальные файлы**. Блочные специальные файлы используются для моделирования устройств, состоящих из набора произвольно адресуемых блоков, таких как диски. Открывая блочный специальный файл и читая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку на устройстве, без обращения к содержащейся на нем файловой системе. Таким же образом символьные специальные файлы используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. По соглашению специальные файлы хранятся в каталоге */dev*. Например, */dev/lp* может быть строковым принтером.

И последнее понятие, которое мы обсудим во введении, — это каналы (pipe), имеющие отношение и к процессам и к файлам. **Канал** (также иногда называемый трубой) представляет собой псевдофайл, который можно использовать для связи двух процессов, как показано на рис. 1.16. Если процессы *A* и *B* захотят пообщаться с помощью канала, они должны установить его заранее. Когда процесс *A* хочет отправить данные процессу *B*, он пишет их в канал, как если бы это был выходной файл. Процесс *B* может прочесть данные, читая их из канала, как если бы он был

файлом с входными данными. Таким образом, соединение между процессами в UNIX выглядит очень похожим на обычное чтение и запись файлов. Более того, только сделав специальный системный вызов, процесс может обнаружить, что выходной файл, в который он пишет данные, не реальный файл, а канал. Файловые системы очень важны. Мы расскажем о них значительно больше в главе 6, а также в главах 10 и 11.



Рис. 1.16. Два процесса, соединенные каналом

Безопасность

Компьютеры содержат большое количество информации, конфиденциальность которой пользователи зачастую хотят сохранить: электронную почту, бизнес-планы и многое другое. В задачу операционной системы входит управление системой защиты подобных файлов, так чтобы они, например, были доступны только пользователям, имеющим на это права.

В качестве простейшего примера, дающего представление о том, как работает система безопасности, рассмотрим систему UNIX. В UNIX для защиты файлов им присваивается 9-битовый двоичный код. Этот код защиты состоит из трех полей по три бита; одно для владельца, второе для других членов группы владельца (пользователи разделяются на группы системным администратором) и третье — для всех остальных. В каждом поле есть бит, определяющий доступ для чтения, бит, определяющий доступ для записи, и бит, разрешающий выполнение. Эти три бита называются **rwx-битами** (read, write, execute). Например, код защиты *rwxr-x--x* означает, что владелец файла может читать, писать или выполнять файл, другие члены группы могут читать или выполнять файл (но не писать в него), а остальные могут только выполнять файл (но не читать или писать). Для каталога *x* означает разрешение на поиск. Дефис означает, что соответствующее разрешение отсутствует.

Кроме защиты файлов, существует еще множество других вопросов безопасности: защита системы от нежелательных гостей, людей, и не только (вирусов). Мы обсудим различные вопросы, связанные с безопасностью, в главе 9.

Оболочка

Операционная система представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики и командные интерпретаторы не являются частью операционной системы, несмотря на их большую важность и полезность. Поскольку есть риск запутаться в этих вещах, в данном разделе мы кратко рассмотрим только командный интерпретатор UNIX, называемый **оболочкой** (shell). Хотя она не входит в операционную систему, но во всю пользуется многими функциями операционной системы и поэтому является хорошим примером того, как могут применяться системные вызовы. Кроме этого,

оболочка предоставляет основной интерфейс между пользователем, сидящим за своим терминалом, и операционной системой, если, конечно, пользователь не использует графический интерфейс. Существует множество оболочек, например *sh*, *csh*, *ksh* и *bash*. Все они поддерживают описанные ниже функции, поскольку произошли от первоначальной оболочки (*sh*).

Когда какой-либо пользователь входит в систему, запускается оболочка. Стандартным входным и выходным устройством для оболочки является терминал (монитор с клавиатурой). Оболочка начинает работу с печати **приглашения** (prompt) — знака доллара, говорящего пользователю, что оболочка ожидает ввода команды. Если теперь пользователь напечатает, например,

```
date
```

оболочка создаст дочерний процесс и запустит программу *date*. Пока дочерний процесс работает, оболочка ожидает его завершения. После завершения дочернего процесса оболочка опять печатает приглашение и пытается прочесть следующую входную строку. Пользователь может перенаправить стандартный вывод данных в файл:

```
date >file
```

Таким же образом можно переопределить устройство, с которого читаются входные данные, как показано ниже:

```
sort <file1 >file2
```

Эта команда предписывает программе сортировки считать данные из файла 1 и вывести результат в файл 2.

Выходные данные одной программы можно использовать в качестве входных данных для другой, соединив их каналом. Так, команда

```
cat file1 file2 file3 | sort >/dev/lp
```

предписывает программе *cat* объединить (con*cat*enate) три файла и послать выходные данные программе *sort*, которая расставит все строки в алфавитном порядке. Результат работы *sort* перенаправляется в файл */dev/lp*, обычно обозначающий принтер.

Если пользователь наберет знак & после команды, оболочка не будет ждать окончания ее выполнения. В этом случае она немедленно напишет новое приглашение. То есть в результате команды

```
cat file1 file2 file3 | sort >/dev/lp &
```

сортировка запустится как фоновое задание, разрешая пользователю продолжать нормальную работу во время выполнения сортировки. Оболочка имеет множество других интересных особенностей, для обсуждения которых у нас здесь, к сожалению, недостаточно места. Но большинство книг по UNIX описывают оболочки довольно подробно [179, 185, 232, 245, 278].

Повторное использование идей

Кибернетика (наука о компьютерах), как и множество других областей знания, находится в сильной зависимости от технологий. Причиной отсутствия автомобилей у древних римлян являлось вовсе не то, что они очень любили ходить пеш-

ком. Машин не было потому, что римляне просто не знали, как их сконструировать. И персональные компьютеры существуют *не* потому, что миллионы людей долгое время хотели иметь свой собственный компьютер, но сдерживали это желание, а потому, что теперь можно относительно дешево их производить. Мы часто забываем, как сильно влияет технология на наше видение систем, и действительно полезно поразмышлять об этом время от времени.

Часто случается, что из-за изменений в технологии некоторые идеи устаревают. Но другие изменения в технологии могут вновь оживить их. Такое случается главным образом тогда, когда происходящие изменения имеют отношение к относительной производительности различных частей системы. Например, когда скорость центрального процессора начинает намного превышать быстродействие памяти, кэш становится очень важной деталью, увеличивающей скорость «медленной» памяти. Если новые технологии в области памяти когда-нибудь создадут память намного более быструю, чем процессор, кэш станет не нужным. Но если затем процессоры опять станут более быстрыми, чем память, кэш появится снова. В биологии вымирание происходит навсегда, но в кибернетике иногда это бывает только на несколько лет.

Из-за такого непостоянства в данной книге время от времени мы будем рассматривать «устаревшие» концепции, то есть идеи, не оптимальные для современных технологий. Но изменения в технологии могут вернуть к жизни некоторые из так называемых «устаревших понятий». По этой причине важно понять, почему концепция является устаревшей и какие изменения в окружающей обстановке могут оживить ее.

Чтобы пояснить нашу точку зрения, рассмотрим несколько примеров. Ранние компьютеры имели вмонтированный в аппаратуру набор команд. Затем появилось микропрограммирование, при котором интерпретатор выполнял команды программно. Аппаратное выполнение устарело. После этого были созданы RISC-компьютеры, и микропрограммирование (то есть интерпретируемое выполнение) тоже стало устаревшим понятием, поскольку исполнение команд напрямую оказалось быстрее. Теперь мы наблюдаем возрождение интерпретации в форме апплетов Java, которые передаются по Интернету и интерпретируются по прибытии. Здесь скорость выполнения не играет решающей роли, поскольку задержки в сети настолько велики, что основное время тратится на них. Но все это тоже однажды может измениться.

Ранние компьютерные системы размещали файлы на диске, располагая их в соседних секторах, один за другим. Хотя эта схема осуществляется очень просто, она не является гибкой, поскольку если файл увеличился в размере, уже не будет места для его хранения. Концепция непрерывного размещения файлов была отвергнута и стала устаревшей. До тех пор, пока не появились компакт-диски. Для них не существует проблемы роста файлов. Внезапно простота непрерывного размещения файлов оказалась гениальной идеей, и на ней сейчас базируются файловые системы компакт-дисков.

И наконец, рассмотрим динамическое связывание. Система MULTICS проектировалась так, чтобы она могла функционировать днем и ночью без остановок. Чтобы программно исправлять системные ошибки, необходимо было найти способ, позволяющий заменять библиотечные процедуры во время их использования.

Для этой цели придумали понятие динамического связывания. После того как система MULTICS отжила свое, это понятие было на время забыто. Но его открыли заново, когда современным операционным системам понадобился способ, позволяющий нескольким программам делить между собой одну библиотечную процедуру, не создавая для себя собственной копии (потому что графические библиотеки выросли до невероятных размеров). Сейчас большинство систем снова поддерживает некоторую форму динамического связывания. Список можно еще продолжить, но мораль описанных выше примеров такова: идея, которая сегодня является устаревшей, завтра может стать гвоздем сезона.

Не только технологии влияют на системы и программное обеспечение. Важную роль играет и экономика. В 60-х и 70-х годах большинство терминалов было механически печатающими устройствами или алфавитно-цифровыми дисплеями с электронно-лучевыми трубками, предназначенным для вывода 25 × 80 символов, а не графическими терминалами с растровым отображением. Этот выбор был обусловлен не технологиями. Растровые графические терминалы использовались еще до 1960 года. Но они стоили несколько десятков тысяч долларов каждый. Только после сильного падения цен люди (а не только военные) смоли задуматься о предоставлении каждому пользователю собственного терминала.

Системные вызовы

Интерфейс между операционной системой и программами пользователя определяется набором системных вызовов, предоставляемых операционной системой. Чтобы на самом деле понять, что же делает операционная система, мы должны подробно рассмотреть этот интерфейс. Системные вызовы, доступные в интерфейсе, меняются от одной операционной системы к другой (хотя лежащая в их основе концепция практически одинакова).

Теперь мы столкнулись с проблемой выбора между (1) неопределенными обобщениями («операционные системы имеют системные вызовы для чтения файлов») и (2) какой-либо конкретной системой («в UNIX существует системный вызов для чтения с тремя параметрами: один для задания файла, второй — для того, чтобы указать, куда нужно поместить прочитанные данные, третий задает количество байтов, которое нужно прочитать»).

Мы выбрали второй подход. При этом способе нужно проделать больше работы, но он обеспечивает лучшее понимание того, что в реальности происходит в операционной системе. Несмотря на то что это обсуждение затрагивает конкретно стандарт POSIX (международный стандарт 9945-1), а, следовательно, также и операционные системы UNIX, System V, BSD, Linux, MINIX и т. д., у большинства других современных операционных систем есть системные вызовы, выполняющие те же самые функции, хотя детали могут быть различны. Так как фактический механизм обращения к системным функциям является в высокой степени машинно-зависимым и часто должен реализовываться на ассемблере, существуют библиотеки процедур, делающие возможным обращение к системным процедурам из программ на C и других языках с тем же успехом.

Очень полезно всегда помнить следующее. Любой компьютер с одним процессором в каждый конкретный момент времени может выполнить только одну команду. Если процесс выполняет программу пользователя в пользовательском режиме и нуждается в системной службе, например чтении данных из файла, он должен выполнить прерывание или команду системного вызова для передачи управления операционной системе. Затем операционная система по параметрам вызова определяет, что требуется вызывающему процессу. После этого она обрабатывает системный вызов и возвращает управление команде, следующей за системным вызовом. В известном смысле выполнение системного вызова похоже на осуществление вызова процедуры, только первый проникает в ядро, а второй этого не делает.

Для того чтобы прояснить механизм системных вызовов, кратко рассмотрим системный вызов read. Как упоминалось выше, у него есть три параметра: первый служит для задания файла, второй указывает на буфер, третий задает количество байтов, которое нужно прочитать. Как практически все системные вызовы, он запускается из программы на C с помощью вызова библиотечной процедуры с тем же именем, что и системный вызов: *read*. Вызов из программы на C может выглядеть так:

```
count = read(fd, buffer, nbytes);
```

Системный вызов (и библиотечная процедура) возвращает количество действительно прочитанных байтов в переменной *count*. Обычно эта величина совпадает с параметром *nbytes*, но может быть меньше, если, например, в процессе чтения процедуре встретился конец файла.

Если системный вызов не может быть выполнен или из-за неправильных параметров или из-за дисковой ошибки, значение счетчика *count* устанавливается равным –1, а номер ошибки помещается в глобальную переменную *errno*. Программы всегда должны проверять результат системного вызова, чтобы отслеживать появление ошибки.

Системные вызовы выполняются за серию шагов. Вернемся к упоминавшемуся выше примеру вызова read для того, чтобы разъяснить этот момент. Сначала при подготовке к вызову библиотечной процедуры read, которая фактически осуществляет системный вызов read, вызывающая программа помещает параметры в стек, как показано в шагах *1–3* на рис. 1.17. Компиляторы C и C++ помещают параметры в стек в обратном порядке, так исторически сложилось (чтобы первым был параметр для *printf*, то есть строка формата оказалась на вершине стека). Первый и третий параметры передаются по значению, а второй параметр передается по ссылке, то есть передается адрес буфера (на то, что это ссылка, указывает символ &), а не его содержимое. Затем следует собственно вызов библиотечной процедуры (шаг *4*). Эта команда процессора представляет собой обычную команду вызова процедуры и применяется для вызова любых процедур.

Библиотечная процедура, возможно, написанная на ассемблере, обычно помещает номер системного вызова туда, где его ожидает операционная система, например в регистр (шаг *5*). Затем она выполняет команду TRAP (эмулированное прерывание) для переключения из пользовательского режима в режим ядра и начинает выполнение с фиксированного адреса внутри ядра (шаг *6*). Запускаемая

программа ядра проверяет номер системного вызова и затем отправляет его нужному обработчику, как правило, используя таблицу указателей на обработчики системных вызовов, индексированную по номерам вызовов (шаг *7*). В этом месте начинает функционировать обработчик системных вызовов (шаг *8*). Как только он завершает свою работу, управление может возвращаться в пространство пользователя к библиотечной процедуре, к команде, следующей за командой TRAP (шаг *9*). Эта процедура в свою очередь передает управление программе пользователя обычным способом, которым производится возврат из вызванной процедуры (шаг *10*).

Чтобы закончить работу, программа пользователя должна очистить стек, как это делается и после каждого вызова процедуры (шаг *11*). Учитывая, что стек растет вниз, последняя команда увеличивает указатель стека ровно настолько, насколько нужно для удаления параметров, помещенных в стек перед запросом *read*. Теперь программа может продолжать свою работу.

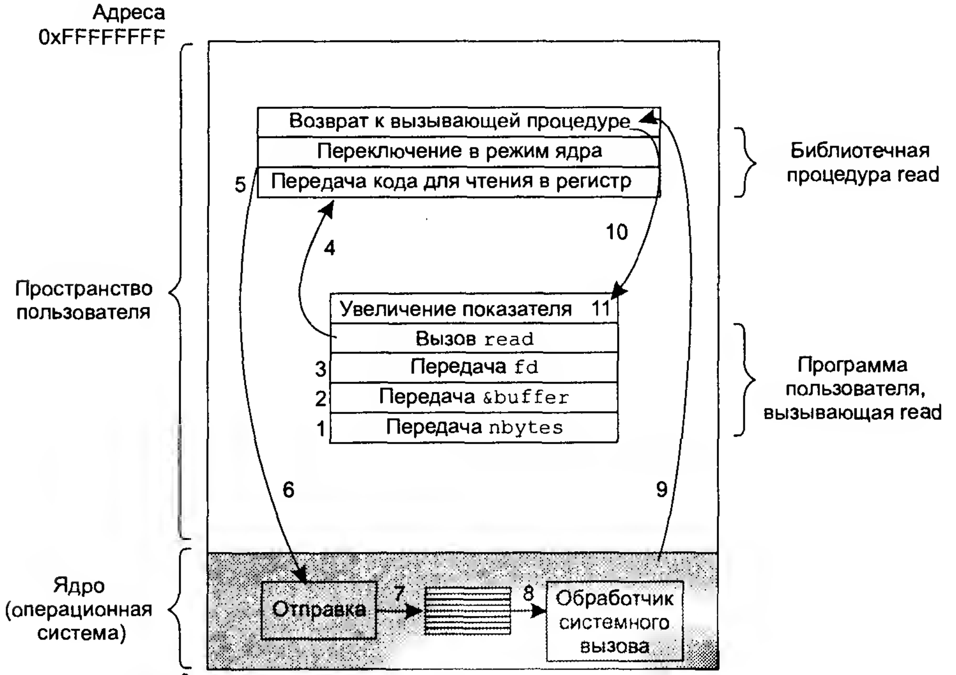


Рис. 1.17. 11 этапов выполнения системного вызова read(fd, buffer, nbytes)

На шаге *9* мы использовали выражение «может возвращаться в пространство пользователя к библиотечной процедуре...» не просто так. Системный вызов может блокировать вызвавшую его процедуру, препятствуя продолжению ее работы. Например, если она пытается прочесть что-то с клавиатуры, а там еще ничего не набрано, процедура должна быть блокирована. В этом случае операционная система ищет процесс, который может быть запущен следующим. Позже, когда нужное устройство станет доступно, система вспомнит о блокированном процессе и шаги *9–11* будут выполнены.

В следующих разделах мы рассмотрим некоторые из наиболее часто применяющихся системных вызовов стандарта POSIX или, точнее, библиотечных процедур, которые выполняют эти вызовы. В POSIX существует более 100 процедурных вызовов. Часть наиболее важных процедурных вызовов представлена в табл. 1.1, где они для удобства распределены в четыре группы. Далее мы кратко опишем каждый вызов и его действие. Службы, предоставляемые этими вызовами, в значительной степени определяют действия операционной системы, так как управление ресурсами на персональном компьютере минимально (по крайней мере, по сравнению с большими машинами, на которых работают несколько пользователей). К этим службам относятся такие функции, как создание и завершение процессов, создание, удаление, чтение и запись файлов, управление каталогами, выполнение ввода и вывода.

Таблица 1.1. Некоторые из основных системных вызовов POSIX

Вызов	Описание
Управление процессами	
Pid=fork()[1]	Создает дочерний процесс, идентичный родительскому
Pid=waitpid(pid, &statloc, options)	Ожидает завершения дочернего процесса
s=execve(name, argv, environp)	Перемещает образ памяти процесса
Exit(status)	Завершает выполнение процесса и возвращает статус
Управление файлами	
fd=open(file, how, ...)	Открывает файл для чтения, записи или того и другого
s=close(fd)	Закрывает открытый файл
n=read(fd, buffer, nbytes)	Читает данные из файла в буфер
n=write(fd, buffer, nbytes)	Пишет данные из буфера в файл
Position=lseek(fd, offset, whence)	Передвигает указатель файла
s=stat(name, &buf)	Получает информацию о состоянии файла
Управление каталогами и файловой системой	
s=mkdir(name, mode)	Создает новый каталог
s=rmdir(name)	Удаляет пустой каталог
s=link(name1, name2)	Создает новый элемент с именем name2, указывающий на namel
s=unlink(name)	Удаляет элемент каталога
s=mount(special, name, flag)	Монтирует файловую систему
s=umount(special)	Демонтирует файловую систему
Разные	
s=chdir(dirname)	Изменяет рабочий каталог
s=chmod(name, mode)	Изменяет биты защиты файла
s=kill(pid, signal)	Посылает сигнал процессу
Seconds=time(&seconds)	Получает время, прошедшее с 1 января 1970 года

[1] Возвращаемая величина s равна –1, если произошла ошибка. Возвращаемые коды выглядят так: *pid* выдает идентификатор процесса, *fd* — описатель файла, *n* — количество байтов, *position* — смещение в файле и *seconds* — прошедшее время. Параметры описываются дальше в тексте.

Особое внимание следует обратить на то, что преобразование вызовов процедур POSIX в системные вызовы не является взаимно однозначным. Стандарт POSIX определяет ряд процедур, которые должны поддерживать совместимые системы, но он не указывает, являются ли они системными вызовами, библиотечными вызовами или чем-нибудь еще. Если процедуру можно выполнить без системного вызова (то есть без переключения в режим работы ядра), то обычно она работает в пространстве пользователя, потому что так быстрее. Однако большинство процедур POSIX выполняет системные вызовы, обычно с одной процедурой, преобразующейся напрямую в системный вызов. В некоторых случаях, особенно когда требуемые процедуры являются всего лишь разновидностями друг друга, один системный вызов обрабатывает сразу несколько библиотечных вызовов.

Системные вызовы для управления процессами

Первая группа в табл. 1.1 управляет процессами. Начнем рассмотрение с вызова fork. Системный вызов fork (разветвление) является единственным способом создания нового процесса в UNIX. Он создает точную копию исходного процесса, включая дескрипторы файла, регистры и т. п. После вызова fork исходный процесс и его копия (родительский и дочерний) развиваются по отдельности друг от друга. Все переменные имеют одинаковые величины во время вызова fork, но как только родительские данные скопированы для создания дочернего процесса, последующие изменения в одном из них уже не влияют на другой. (Текст программы, который не изменяется, распределяется между родительским и дочерним процессами). Вызов fork возвращает величину, равную нулю в дочернем процессе и равную идентификатору дочернего процесса или **PID** в родительском. Используя возвращенный PID, два процесса могут различить, какой из них родительский, а какой — дочерний.

В большинстве случаев после вызова fork дочернему процессу необходимо выполнить программный код, отличный от предназначенного для родительского процесса. Рассмотрим пример оболочки. Она читает команды с терминала, запускает дочерний процесс, ждет, пока дочерний процесс выполнит команду, и читает следующую команду после завершения работы дочернего процесса. Ожидая, пока дочерний процесс закончит работу, родительский процесс выполняет системный вызов waitpid, который ожидает завершения дочернего процесса (или всех дочерних процессов, если их на данный момент несколько). Waitpid может ждать окончания какого-либо определенного дочернего процесса или любого дочернего процесса, для этого нужно задать первый параметр вызова равным –1. Когда waitpid выполнен, указатель, задаваемый вторым параметром statloc, будет установлен на статус завершения дочернего процесса (нормальное или аварийное завершение и выходное значение). Третий параметр определяет различные необязательные настройки.

Теперь рассмотрим, как вызов fork используется оболочкой. Когда печатается команда, оболочка создает дочерний процесс, который должен выполнить команду пользователя. Он делает это с помощью системного вызова execve, заменяющего весь его образ памяти файлом, названным в первом параметре. (Фактически самим системным вызовом является exec, но несколько различных библиотечных

процедур вызывают его с разными параметрами и незначительно отличающимися именами. Мы здесь воспользуемся ими как системными вызовами.) Весьма упрощенная оболочка, иллюстрирующая использование команд fork, waitpid и execve показана в листинге 1.1.

Листинг 1.1. Усеченная оболочка[1]

```
#define TRUE 1
while (TRUE) {                                /* вечный цикл */
       type_prompt( );                        /* печать приглашения на экране */
       read_command(command, parameters);     /* читать входные данные с терминала */
       if (fork( ) != 0) {                    /*запускает дочерний процесс */
           /* текст родительского процесса */
           waitpid(-1, &status, 0);           /* ждать окончания дочернего процесса */
       } else {
           /* текст дочернего процесса */
           execve(command, parameters, 0);    /* выполнение command */
       }
}
```

В самом общем случае у команды execve есть три параметра: имя файла, который будет выполняться, указатель на массив аргументов и указатель на массив переменных окружения. Эти параметры мы кратко обсудим в дальнейшем. Различные библиотечные программы, включая *execl*, *execv*, *execle* и *execve*, разрешают пропускать параметры или определять их другими способами. В книге мы воспользуемся названием exec для того, чтобы представить системный вызов, вызываемый всеми этими процедурами.

Рассмотрим следующую команду:

```
cp file1 file2
```

которая используется для копирования файла *file1* в файл *file2*. После создания оболочкой дочернего процесса последний находит и исполняет файл *cp* и передает ему имена исходного и целевого файлов.

Основной модуль программы *cp* (как и большинство других головных программ на С) содержит определение:

```
main(argc, argv, envp)
```

в котором в параметр *argc* входит количество записей в командной строке, включая имя программы. Например, для строки вверху *argc* равен 3.

Второй параметр *argv* является указателем на массив указателей. Элемент *i* массива указывает на *i*-ю запись в командной строке. В нашем примере *argv*[0] должен указывать на слово «cp», а *argv*[1] и *argv*[2] — на слова «file1» и «file2» соответственно.

Третий параметр функции *main*, *envp*, является указателем на массив строковых переменных окружения вида имя=величина, которые используются для передачи программе такой информации, как тип терминала или имя домашнего каталога. В листинге 1.1 третий параметр равен нулю, поскольку ничего не передается дочернему процессу.

[1] На протяжении всей книги значение константы *TRUE* предполагается равным 1. — *Примеч. авт.*

Если команда exec кажется сложной, не огорчайтесь, потому что это один из наиболее сложных системных вызовов в POSIX. Все остальные намного проще. В качестве еще одного примера рассмотрим exit, процессы должны использовать его при завершении работы. У него есть всего один параметр, статус выхода, изменяющийся от 0 до 255. Он возвращается родительскому процессу через параметр *statloc* в системном вызове waitpid.

В UNIX под процессы отводится часть памяти, которая, в свою очередь, делится на три сегмента: **текстовый** (то есть код программы), **сегмент данных** (переменные) и **сегмент стека**. Сегмент данных растет снизу вверх, а стек увеличивается сверху вниз, как показано на рис 1.18. Между ними существует часть неиспользованного адресного пространства. Стек автоматически занимает такую часть этого участка памяти, какую необходимо, но расширение сегмента данных выполняется явным образом. Для этого используется специальный системный вызов brk, задающий новый адрес для границы сегмента данных. Однако этот вызов не определен стандартами POSIX, так как программистам для динамического распределения памяти рекомендуется использовать библиотечную процедуру *malloc*. Было решено, что низкоуровневую реализацию процедуры *malloc* не следует стандартизировать, потому что мало кто вызывает ее напрямую.

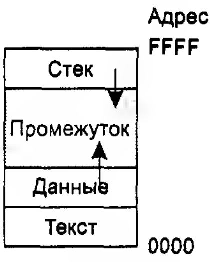


Рис. 1.18. Под процессы отводится три сегмента: текст, данные и стек

Системные вызовы для управления файлами

Многие системные вызовы имеют отношение к файловой системе. В этом разделе мы рассмотрим вызовы, работающие с отдельными файлами, а в следующем разделе обратимся к тем, которые оперируют каталогами или файловой системой в целом.

Чтобы прочитать или записать файл, его сначала нужно открыть при помощи вызова open. Для этого вызова указывается имя открываемого файла (задается или абсолютный путь файла, или ссылка на рабочий каталог) и код *O_RDONLY*, *O_WRONLY* или *O_RDWR*, означающий, что файл открывается для чтения, записи или и того и другого. Для создания нового файла используется код O_*CREAT*. Возвращаемый дескриптор файла затем можно употребить при чтении или записи. Потом файл закрывается с помощью вызова close, который делает дескриптор файла доступным при следующем открытии (open).

Наиболее часто используемыми вызовами, без сомнения, являются read и write. Вызов read мы уже обсуждали, write имеет те же самые параметры.

Несмотря на то что большинство программ читает и записывает файлы с помощью последовательного доступа, некоторым прикладным программам необхо-

дима возможность доступа к любой, случайно выбранной части файла. Связанный с каждым файлом указатель содержит текущую позицию в файле. Когда чтение (запись) осуществляется последовательно, он обычно указывает на байт, который должен быть прочитан (записан) следующим. Вызов `lseek` может изменить значение позиции указателя, так что следующий вызов `read` или `write` начнет операцию где-либо в другой части файла.

У вызова `lseek` есть три параметра: первый — это идентификатор файла, второй — позиция в файле, а третий говорит, является ли второй параметр позицией в файле относительно начала файла (абсолютная позиция), относительно текущей позиции или относительно конца файла. Вызов `lseek` возвращает абсолютную позицию в файле после изменения указателя.

Для каждого файла UNIX хранит следующие данные: тип файла (обычный, специальный, каталог и т. д.), размер, время последнего изменения и другую информацию. Программа может запросить эту информацию через системный вызов `stat`. Его первый параметр определяет требуемый файл, а второй указывает на структуру, куда нужно поместить информацию.

Системные вызовы для управления каталогами

В этом разделе мы рассмотрим некоторые системные вызовы, относящиеся скорее к каталогам и файловой системе в целом, нежели просто к определенному файлу, как в предыдущем разделе. Первые два вызова, `mkdir` и `rmdir`, соответственно создают и удаляют пустые каталоги. Следующий вызов — `link`. Он разрешает одному файлу появляться под двумя или более именами, часто в разных каталогах. Этот вызов обычно используется, когда несколько программистов, работающих в одной команде, должны совместно использовать один общий файл. Тогда этот файл может появиться в каталоге у каждого из программистов, возможно, под другим именем. Разделение (совместное использование) файла — это не то же самое, что копирование файла для каждого члена команды. При разделении файла изменения, производимые одним программистом, немедленно становятся видимыми для остальных — все происходит в одном файле. А при создании копии файла последующие изменения не влияют на другие копии этого файла.

Чтобы увидеть, как работает вызов `link`, рассмотрим ситуацию на рис. 1.19, *а*. Два пользователя, *ast* и *jim*, имеют свои собственные каталоги ast и jim с файлами. Если теперь пользователь *ast* запустит программу, содержащую системный вызов

```
link("/usr/jim/memo", "/usr/ast/note");
```

то файл *memo* в каталоге Джима появится в каталоге Аста под названием *note*. Соответственно, */usr/jim/memo* и */usr/ast/note* теперь будут ссылаться на один и тот же файл. Хранятся ли каталоги пользователей в каталоге */usr*, */user*, */home* или где-либо еще, определяется локальным системным администратором.

Возможно, станет понятнее, что делает системный вызов `link`, если разобраться в том, как он работает. Каждый файл в UNIX имеет уникальный номер — свой i-номер, который идентифицирует файл (identification — идентификация). I-номер — это индекс в таблице **i-узлов** (i-nodes), содержащей по одному на файл. Каждый i-узел включает в себя информацию о хозяине файла, о том, какие блоки на диске

он занимает и т. д. Каталог представляет собой просто файл, содержащий набор пар (i-номер, ASCII-имя). В первой версии UNIX под каждый элемент каталога было отведено 16 байт: два байта для i-номера и 14 байт для названия. Хотя теперь для поддержки длинных имен используется более сложная структура, концептуально каталог все еще остается набором пар (i-номер, ASCII-имя). На рис. 1.19 файл *mail* имеет i-номер, равный 16 и т. д. Вызов `link` просто создает новый элемент каталога, возможно, с новым именем, используя i-номер существующего файла. На рис. 1.19, *б* два элемента имеют одинаковый i-номер (70) и, таким образом, ссылаются на один и тот же файл. Если впоследствии один из них будет удален с помощью системного вызова `unlink`, другой элемент останется. Если будут удалены оба файла, UNIX убедится, что больше нет записей, соответствующих этому файлу (поле в таблице i-узлов хранит данные с номером элемента каталога, указывающие на файл), и удалит файл с диска.



а *б*

Рис. 1.19. Два каталога до присоединения /usr/jim/memo к каталогу ast (*а*); те же каталоги после вызова link (*б*)

Как упоминалось выше, системный вызов `mount` позволяет объединять в одну две файловые системы. Обычная ситуация такова: на жестком диске находится корневая файловая система, содержащая двоичные (исполняемые) версии общих команд и наиболее часто использующиеся файлы. При этом пользователь может вставить в дисковод гибкий диск для чтения файлов.

При помощи системного вызова `mount` файловую систему с гибкого диска можно присоединить к корневой файловой системе, как показано на рис. 1.20. Типичный оператор на языке C, выполняющий монтирование, выглядит так:

```
mount("/dev/fd0", "/mnt", 0);
```

где первым параметром является имя специального блочного файла на диске 0, второй параметр — это место в дереве, куда будет вмонтирована файловая система, а третий параметр говорит о том, монтируется ли встроенная файловая система для чтения и записи или только для чтения.

а

/usr/ast

16	mail
81	games
40	test

/usr/jim

31	bin
70	memo
59	f.c.
38	prog1

б

/usr/ast

16	mail
81	games
40	test
70	note

/usr/jim

31	bin
70	memo
59	f.c.
38	prog1

Рис. 1.20. Файловая система: до вызова mount (*а*); после вызова mount (*б*)

После вызова mount доступ к файлу на диске 0 можно получить, просто указав его путь из корневого или рабочего каталога, независимо от того, на каком диске он находится. В действительности второй, третий и четвертый диски тоже можно встроить в любое удобное место в дереве. Вызов mount позволяет объединить съемные носители в единую интегрированную файловую структуру, не заботясь о том, на каком из устройств фактически находится файл. Хотя в нашем примере рассматривались гибкие диски, жесткие диски или их части (часто называемые **разделами** (partition) или **младшими устройствами** (minor devices)) монтируются аналогично. Когда файловая система более не нужна, ее можно демонтировать с помощью системного вызова umount.

Разные системные вызовы

Существует, помимо описанных выше, еще множество системных вызовов. Сейчас мы рассмотрим только четыре из них. Вызов chdir изменяет текущий рабочий каталог. После вызова

```
chdir("/usr/ast/test");
```

процедура открытия файла *xyz* откроет */usr/ast/test/xyz*. Использование понятия рабочего каталога избавляет от необходимости постоянно набирать длинные абсолютные пути файлов.

В UNIX для каждого файла определен режимный код файла, иногда также называемый модой (mode), используемый для его защиты. Режимный код включает в себя биты чтения-записи-выполнения для владельцев, группы и других пользователей. Системный вызов chmod предоставляет возможность изменения режимного кода файла. Например, следующий вызов предоставит всем, кроме владельца, доступ к файлу только для чтения; владелец же сможет еще и выполнять файл:

```
chmod("file", 0644);
```

Системный вызов kill позволяет пользователям и пользовательским процессам посылать сигналы. Если процесс готов принять определенный сигнал, то при его прибытии запускается обработчик сигналов. Если же процесс не готов обработать сигнал, тогда прибытие сигнала уничтожает процесс (отсюда произошло название вызова, kill в переводе означает «убивать, уничтожать»).

Стандартом POSIX определено несколько процедур, имеющих отношение ко времени. Например, time возвращает текущее время в секундах, причем нулем считается полночь 1 января 1970 года (имеется в виду начало дня, а не его конец). На компьютерах с 32-разрядными словами максимальная величина, которую может вернуть time, составляет $2^{32}-1$ с (используется положительное целое число). Эта величина соответствует периоду немногим более 136 лет. Таким образом, в 2106 году 32-разрядные системы UNIX «сойдут с ума», имитируя знаменитую проблему 2000 года (Y2K). Если сейчас у вас 32-разрядная система UNIX, мы рекомендуем вам поменять ее на 64-разрядную, прежде чем наступит 2106 год.

Windows Win32 API

До сих пор мы концентрировали свое внимание в первую очередь на системе UNIX. Теперь самое время обратить внимание на Windows. В Windows и UNIX

фундаментально отличаются соответствующие модели программирования. Программы UNIX состоят из кода, который выполняет те или иные действия, обращаясь к системе с системными запросами для предоставления ему конкретных услуг. В противоположность этому программы в Windows обычно приводятся в действие событиями. Основной модуль программы ждет, когда произойдет какое-либо событие, затем вызывает процедуру для его обработки. Типичными событиями являются: нажатие клавиши мыши или клавиатуры, передвижение мыши или появление гибкого диска в дисководе. Затем обработчики, вызываемые для обработки события, переписывают содержимое экрана и внутреннее состояние программы. Все это ведет к совершенно отличному от UNIX стилю программирования, но поскольку наша книга посвящена функциям и структуре операционной системы, различные модели программирования не имеют к нам сейчас особого отношения.

Конечно, в Windows тоже есть системные вызовы. В UNIX вызовы почти один к одному идентичны библиотечным процедурам (вспомним read), используемым для обращения к системным вызовам. Другими словами, для каждого системного вызова существует одна библиотечная процедура, обычно с тем же названием, которая вызывается для обращения к нему. Это было показано на рис. 1.17. Кроме того, в стандарте POSIX всего лишь около 100 процедурных вызовов.

Ситуация в системе Windows совершенно иная. Во-первых, фактические системные вызовы и запускающиеся для их выполнения библиотечные вызовы полностью разделены. Корпорацией Microsoft определен набор процедур, называемый **Win32 API** (Application Program Interface — интерфейс прикладных программ). Предполагается, что программисты должны использовать его для вызова служб операционной системы. Этот интерфейс частично поддерживается всеми версиями Windows, начиная с Windows 95. Отделяя интерфейс от фактических системных вызовов, Microsoft поддерживает возможность изменения со временем действительных системных вызовов (даже от одной версии к другой), не делая при этом недействительными существующие программы. Ситуация на самом деле складывается неоднозначная, так как в Windows 2000 появилось много новых вызовов, ранее недоступных. В этом разделе Win32 означает интерфейс, поддерживаемый всеми версиями Windows.

Количество вызовов в Win32 API крайне велико, вызовы исчисляются тысячами. Кроме того, хотя многие из них являются системными, существенное число работает целиком в пространстве пользователя. Из-за этого в Windows невозможно понять, является ли вызов системным (то есть выполняемым ядром), или просто библиотечным вызовом, который обрабатывается в пространстве пользователя. В принципе вызов, считающийся системным в одной версии Windows, может выполняться в пользовательском пространстве в других версиях, и наоборот. При обсуждении системных вызовов Windows в этой книге мы будем использовать соответствующие процедуры Win32, поскольку Microsoft гарантирует, что они не будут меняться. Но стоит всегда помнить, что не все они являются настоящими системными вызовами (то есть прерываниями с переключением в режим ядра).

Другое различие заключается в том, что в UNIX графический интерфейс пользователя (например, X Windows или Motif) запускается целиком в пользовательском пространстве. Поэтому для вывода на экран достаточно системного вызова

write и несколько других незначительных вызовов. Конечно, существует достаточно большое количество обращений к X Windows и GUI, но они не являются системными вызовами в любом смысле этого слова.

В противоположность этому Win32 API имеет огромное количество вызовов для управления окнами, геометрическими фигурами, текстом, шрифтами, полосами прокрутки, диалоговыми окнами, пунктами меню и другими элементами графического интерфейса. В том случае, когда графическая подсистема запускается в режиме ядра (это верно для большинства версий Windows, но не для всех), вызовы являются системными; в противном случае вызовы являются только библиотечными. Должны ли мы обсуждать эти вызовы в книге или нет? Так как они на самом деле не связаны с функциями операционной системы, мы решили этого не делать, несмотря на то, что они выполняются ядром. Читателям, интересующимся Win32 API, мы рекомендуем обратиться к любой из множества книг по этому предмету, например [146, 271 или 302].

Даже перечисление всех вызовов Win32 API выходит за рамки этой книги, поэтому мы ограничимся теми из них, которые приблизительно соответствуют функциональности вызовов UNIX, перечисленных в табл. 1.1. Эти вызовы показаны в табл. 1.2.

Таблица 1.2. Вызовы Win32 API, приблизительно соответствующие вызовам UNIX, перечисленным в табл. 1.1

UNIX	Win32	Описание
Fork	CreateProcess	Создать новый процесс
Waitpid	WaitForSingleObject	Ждать завершения процесса
Execve	(нет)	CreateProcess=fork+execve
Exit	ExitProcess	Завершить выполнение
Open	CreateFile	Создать файл или открыть существующий файл
Close	CloseHandle	Закрыть файл
Read	ReadFile	Читать данные из файла
Write	WriteFile	Записать данные в файл
Lseek	SetFilePointer	Передвинуть указатель файла
Stat	GetFileAttributesEx	Получить различные атрибуты файла
Mkdir	CreateDirectory	Создать новый каталог
Rmdir	RemoveDirectory	Удалить пустой каталог
Link	(нет)	Win32 не поддерживает связи
Unlink	DeleteFile	Удалить существующий файл
Mount	(нет)	Win32 не поддерживает монтирование
Umount	(нет)	Win32 не поддерживает монтирование
chdir	SetCurrentDirectory	Изменить текущую рабочую папку
chmod	(нет)	Win32 не поддерживает защиту файла (хотя NT поддерживает)
kill	(нет)	Win32 не поддерживает сигналы
time	GetLocalTime	Получить текущее время

Теперь кратко пройдемся по списку, представленному в табл. 1.2. Вызов CreateProcess создает новый процесс. Он комбинирует работу вызовов fork и execve

в UNIX. У этого вызова есть множество параметров, определяющих свойства созданных процессов. В Windows нет такой иерархии процессов, как в UNIX, поэтому здесь нет концепции родительских и дочерних процессов. После создания нового процесса «создатель» и «созданный» становятся равноправными. WaitForSingleObject используется для ожидания события. Существует множество событий, которые можно ждать. Если параметром указан процесс, тогда вызывающая программа ждет завершения определенного процесса. Завершение процесса выполняется с помощью вызова ExitProcess.

Следующие шесть вызовов оперируют файлами и функционально похожи на свои аналоги в UNIX, несмотря на то что они имеют различные параметры и детали. Таким образом, файлы можно открывать, закрывать и читать практически так же, как в UNIX. Вызовы SetFilePointer и GetFileAttributesEx устанавливают позицию указателя и получают некоторые из атрибутов файла.

В Windows также есть каталоги, их создают и удаляют вызовы CreateDirectory и RemoveDirectory соответственно. Здесь тоже есть понятие текущего каталога, он задается с помощью SetCurrentDirectory. Для получения текущего времени используется GetLocalTime.

В интерфейсе Win32 не существует связанных файлов, монтирования файловых систем, защиты файлов и сигналов, поэтому также нет и вызовов, соответствующих вызовам UNIX. Конечно, в Win32 есть огромное количество самых разных других вызовов, которых не имеет UNIX, особенно относящихся к управлению графическим интерфейсом. Однако система Windows 2000 имеет тщательно продуманную систему безопасности и, кроме того, поддерживает связи между файлами.

Стоит сделать еще одно, последнее замечание относительно Win32. Win32 не является полностью единообразным и последовательным интерфейсом. Причиной этого явилась необходимость обратной совместимости с ранее использовавшимся в Windows 3.x 16-разрядным интерфейсом.

Структура операционной системы

Теперь, когда мы уже видели, как выглядят операционные системы снаружи (то есть мы знакомы с программным интерфейсом), самое время заглянуть внутрь. Чтобы получить представление обо всем спектре возможных вариантов, в следующих разделах мы исследуем пять различных использующихся (или использовавшихся ранее) структур. Исследование это нельзя назвать всесторонним, здесь лишь рассматривается несколько моделей, применявшихся на практике в разных системах. Их пять — монолитные системы, многоуровневые системы, виртуальные машины, экзоядро и модель клиент-сервер.

Монолитные системы

В общем случае организация монолитной системы представляет собой «большой беспорядок». То есть структура как таковая отсутствует. Операционная система

написана в виде набора процедур, каждая из которых может вызывать другие, когда ей это нужно. При использовании такой техники каждая процедура системы имеет строго определенный интерфейс в терминах параметров и результатов, и каждая имеет возможность вызвать любую другую для выполнения некоторой необходимой для нее работы.

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их в единый объектный файл с помощью компоновщика. Здесь, по существу, полностью отсутствует сокрытие деталей реализации — каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которой большая часть информации является локальной для модуля и процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут иметь некоторую структуру. При обращении к системным вызовам, поддерживаемым операционной системой, параметры помещаются в строго определенные места — регистры или стек, после чего выполняется специальная команда прерывания, известная как вызов ядра или вызов супервизора. Эта команда переключает машину из режима пользователя в режим ядра и передает управление операционной системе, что видно на шаге *6* рис. 1.17. Затем операционная система проверяет параметры вызова, чтобы определить, какой системный вызов должен быть выполнен. После этого операционная система обращается к таблице как к массиву с номером системного вызова в качестве индекса. В *k*-м элементе таблицы содержится ссылка на процедуру обработки системного вызова *k* (шаг *7* на рис. 1.17). Такая организация операционной системы предполагает следующую структуру:

1. Главная программа, которая вызывает требуемую служебную процедуру.
2. Набор служебных процедур, выполняющих системные вызовы.
3. Набор утилит, обслуживающих служебные процедуры.

В этой модели для каждого системного вызова имеется одна служебная процедура. Утилиты выполняют функции, которые нужны нескольким служебным процедурам. Деление процедур на три уровня показано на рис. 1.21.

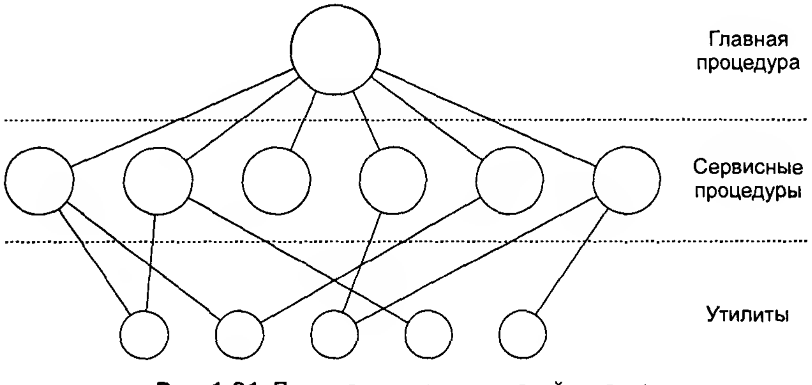


Рис. 1.21. Простая модель монолитной системы

Многоуровневые системы

Обобщением подхода, изображенного на рис. 1.21, является организация операционной системы в виде иерархии уровней. Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven (Нидерланды) Э. Дейкстрой (E. W. Dijkstra) и его студентами в 1968 году. Она была простой пакетной системой для голландского компьютера Electrologica X8, память которого состояла из 32 К 27-разрядных слов. Система включала 6 уровней, как показано в табл. 1.3. Уровень 0 занимался распределением времени процессора, переключая процессы при возникновении прерывания или при срабатывании таймера. Над уровнем 0 система состояла из последовательных процессов, каждый из которых можно было запрограммировать, не заботясь о том, что на одном процессоре запущено несколько процессов. Другими словами, уровень 0 обеспечивал базовую многозадачность процессора.

Таблица 1.3. Структура операционной системы THE

Уровень	Функция
5	Оператор
4	Программы пользователя
3	Управление вводом-выводом
2	Связь оператор-процесс
1	Управление памятью и барабаном
0	Распределение процессора и многозадачность

Уровень 1 управлял памятью. Он выделял процессам пространство в оперативной памяти и на магнитном барабане объемом 512 К слов для тех частей процессов (страниц), которые не помещались в оперативной памяти. Процессы более высоких уровней не заботились о том, находятся ли они в данный момент в памяти или на барабане. Программное обеспечение уровня 1 обеспечивало попадание страниц в оперативную память по мере необходимости.

Уровень 2 управлял связью между консолью оператора и процессами. Таким образом, все процессы выше этого уровня имели свою собственную консоль оператора. Уровень 3 управлял устройствами ввода-вывода и буферизовал потоки информации к ним и от них. Любой процесс выше уровня 3, вместо того чтобы работать с конкретными устройствами, с их разнообразными особенностями, мог обращаться к абстрактным устройствам ввода-вывода, обладающим удобными для пользователя характеристиками. На уровне 4 работали пользовательские программы, которым не надо было заботиться ни о процессах, ни о памяти, ни о консоли, ни об управлении устройствами ввода-вывода. Процесс системного оператора размещался на уровне 5.

Дальнейшее обобщение многоуровневой концепции было сделано в операционной системе MULTICS. В ней уровни представляли собой серию концентрических колец, где внутренние кольца являлись более привилегированными, чем внешние. Когда процедура внешнего кольца хотела вызвать процедуру кольца, лежащего внутри, она должна была выполнить эквивалент системного вызова, то есть коман-

ду TRAP, параметры которой тщательно проверяются перед тем, как выполняется вызов. Хотя операционная система в MULTICS являлась частью адресного пространства каждого пользовательского процесса, аппаратура обеспечивала защиту данных на уровне сегментов памяти, разрешая или запрещая доступ к индивидуальным процедурам (в действительности к сегментам памяти) для записи, чтения или выполнения.

Стоит отметить, что в системе THE многоуровневая схема представляла собой исключительно конструкционное решение и все части системы были, в конечном счете, связаны в один объектный файл, а в MULTICS механизм разделения колец действовал во время исполнения на аппаратном уровне.

Преимущество подхода MULTICS заключается в том, что его можно расширить и на структуру пользовательских подсистем. Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить ее в кольце n, в то время как студенческие программы будут работать в кольце $n + 1$, так что они не смогут изменить свои оценки.

Виртуальные машины

Исходная версия OS/360 была системой исключительно пакетной обработки. Однако множество пользователей OS/360 желали работать в системе с разделением времени, поэтому различные группы программистов как в самой корпорации IBM, так и вне ее решили написать для этой машины системы с разделением времени. Официальная система с разделением времени от IBM, которая называлась TSS/360, поздно вышла в свет и оказалась настолько громоздкой и медленной, что на нее перешли немногие. В конечном счете от нее отказались, но уже после того, как ее разработка потребовала около 50 млн долларов [135]. Группа из научного центра IBM в Кембридже, штат Массачусетс, разработала в корне отличающуюся от нее систему, которую IBM в результате приняла как законченный продукт. Сейчас она широко используется на еще оставшихся мэйнфреймах.

Эта система, в оригинале называвшаяся CP/CMS, а позже переименованная в VM/370 [279], была основана на следующем проницательном наблюдении: система с разделением времени обеспечивает (1) многозадачность и (2) расширенную машину с более удобным интерфейсом, чем тот, что предоставляется оборудованием напрямую. VM/370 основана на полном разделении этих двух функций.

Сердце системы, называемое **монитором виртуальной машины**, работает с оборудованием и обеспечивает многозадачность, предоставляя верхнему слою не одну, а несколько виртуальных машин, как показано на рис. 1.22. Но, в отличие от всех других операционных систем, эти виртуальные машины не являются расширенными. Они не поддерживают файлы и прочие удобства, а представляют собой точные копии голой аппаратуры, включая режимы ядра и пользователя, ввод-вывод данных, прерывания и все остальное, присутствующее на реальном компьютере.

Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каждой из них может работать любая операционная система, которая запускается прямо на аппаратуре. На разных виртуальных машинах могут (а зачастую так и происходит) функционировать различные операционные системы. На некоторых из них для обработки пакетов и транзакций работают потомки OS/360, а на других

для интерактивного разделения времени пользователей работает однопользовательская интерактивная система **CMS** (Conversational Monitor System – система диалоговой обработки).

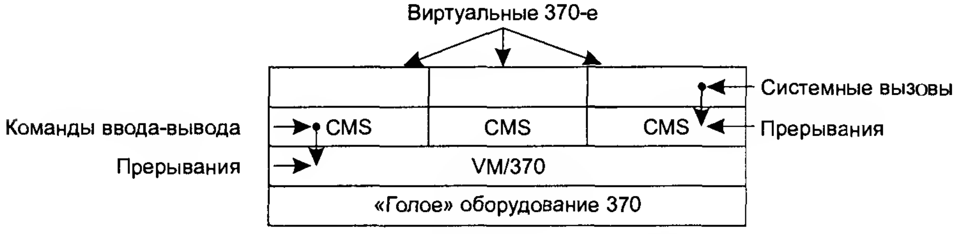


Рис. 1.22. Структура VM/370 с системой CMS

Когда программа операционной системы CMS выполняет системный вызов, он прерывает операционную систему на своей собственной виртуальной машине, а не на VM/370, как произошло бы, если бы он работал на реальной машине вместо виртуальной. Затем CMS выдает обычные команды ввода-вывода для чтения своего виртуального диска или другие команды, которые ей могут понадобиться для выполнения вызова. Эти команды ввода-вывода перехватываются VM/370, которая выполняет их в рамках моделирования реального оборудования. При полном разделении функций многозадачности и предоставления расширенной машины каждая часть может быть намного проще, гибче и удобней для обслуживания.

Идея виртуальной машины очень часто используется в наши дни, но в несколько другом контексте: для работы старых программ, написанных для системы MS-DOS на Pentium (или на других 32-разрядных процессорах Intel). При разработке компьютера Pentium и его программного обеспечения обе компании, Intel и Microsoft, понимали, что возникнет острая потребность в работе старых программ на новом оборудовании. Поэтому корпорация Intel создала на процессоре Pentium режим виртуального процессора 8086. В этом режиме машина действует как 8086 (которая с точки зрения программного обеспечения идентична 8088), включая 16-разрядную адресацию памяти с ограничением объема памяти в 1 Мбайт.

Такой режим используется системой Windows и другими операционными системами для запуска программ MS-DOS. Программы запускаются в режиме виртуального процессора 8086. Пока они выполняют обычные команды, они работают напрямую с оборудованием. Но когда программа пытается обратиться по прерыванию к операционной системе, чтобы сделать системный вызов, или пытается напрямую осуществить ввод-вывод данных, происходит прерывание с переключением на монитор виртуальной машины.

Возможны два варианта устройства. Первый: сама система MS-DOS загружена в адресное пространство виртуальной машины 8086, так что монитор виртуальной машины только отсылает прерывания назад к MS-DOS, как это происходит на реальной 8086. Когда затем MS-DOS пытается самостоятельно осуществить ввод-вывод, операция перехватывается и выполняется монитором виртуальной машины.

В другом варианте монитор виртуальной машины перехватывает первое прерывание и сам выполняет ввод-вывод, так как он знает все системные вызовы

MS-DOS и имеет представление о том, что должно делать каждое прерывание. Этот вариант не столь безупречен, как первый, потому что, в отличие от первого варианта, он корректно моделирует только MS-DOS и никакие другие операционные системы. С другой стороны, он намного быстрее работает, так как избегает проблем запуска MS-DOS для выполнения ввода-вывода. Существует еще один недостаток фактического запуска MS-DOS в режиме виртуальной 8086: MS-DOS очень часто оперирует флагом разрешения/запрещения прерывания, а моделирование этого требует больших затрат.

Стоит отметить, что ни один из двух описанных методов в действительности не является те же самым, чем была VM/370, потому что смоделированная машина представляет собой только 8086, а не полноценный Pentium. В системе VM/370 можно было запустить на виртуальной машине саму VM/370. На Pentium нельзя запустить, скажем, операционную систему Windows на виртуальной 8086, потому что не существует версий Windows, работающих на этой машине. Даже для самых старых версий Windows необходим как минимум 286-й процессор, а моделирование 286 не поддерживается (не говоря уже об эмуляции Pentium). Однако, если немного модифицировать двоичный код Windows, подобная модель становится возможна, и даже возможна ее коммерческая реализация.

Кроме того, виртуальные машины используются, правда, несколько другим способом, для работы программ Java. Когда корпорация Sun Microsystems придумала язык программирования Java, она также разработала виртуальную машину (то есть архитектуру компьютера), называемую **JVM** (Java Virtual Machine — виртуальная машина Java). Компилятор Java выдает код для JVM, который затем обычно выполняется программным интерпретатором JVM. Преимущество этого подхода заключается в том, что код JVM можно передавать через Internet на любой компьютер, имеющий интерпретатор JVM, и запускать там. Если бы компилятор выдавал двоичные программы, например, для компьютеров SPARC или Pentium, их было бы нельзя куда-либо передать и запустить в работу так просто, как это происходит с JVM. (Конечно, компания Sun могла бы разработать компилятор, который выдавал бы двоичные коды SPARC, и затем использовать интерпретатор SPARC, но структура JVM намного проще для интерпретации.) Другое преимущество JVM заключается в том, что когда интерпретатор реализован должным образом, что вовсе не тривиально, приходящие JVM-программы можно проверить в целях безопасности и затем запустить в защищенной среде, так что эти программы не смогут похитить данные или причинить какой-нибудь иной вред.

Экзоядро

В системе VM/370 каждый пользователь получает точную копию настоящей машины. На Pentium, в режиме виртуальной машины 8086, каждый пользователь получает точную копию другой машины. Развив эту идею дальше, исследователи из Массачусетсского технологического института изобрели систему, которая обеспечивает каждого пользователя абсолютной копией реального компьютера, но с подмножеством ресурсов [111]. Например, одна виртуальная машина может получить блоки на диске с номерами от 0 до 1023, следующая — от 1024 до 2047 и т. д.

На нижнем уровне в режиме ядра работает программа, которая называется **экзоядро (exokernel)**. В ее задачу входит распределение ресурсов для виртуальных машин, а после этого проверка их использования (то есть отслеживание попыток машин использовать чужой ресурс). Каждая виртуальная машина на уровне пользователя может работать с собственной операционной системой, как на VM/370 или виртуальных 8086-х для Pentium, с той разницей, что каждая машина ограничена набором ресурсов, которые она запросила и которые ей были предоставлены.

Преимущество схемы экзоядра заключается в том, что она позволяет обойтись без уровня отображения. При других методах работы каждая виртуальная машина считает, что она использует свой собственный диск с нумерацией блоков от 0 до некоторого максимума. Поэтому монитор виртуальной машины должен поддерживать таблицы преобразования адресов на диске (и всех других ресурсов). Необходимость преобразования отпадает при наличии экзоядра, которому нужно только хранить запись о том, какой виртуальной машине выделен данный ресурс. Такой подход имеет еще одно преимущество: он отделяет многозадачность (в экзоядре) от операционной системы пользователя (в пространстве пользователя) с меньшими затратами, так как для этого ему необходимо всего лишь не допускать вмешательства одной виртуальной машины в работу другой.

Модель клиент-сервер

Сис тема VM/370 сильно выигрывает в простоте благодаря переносу значительной части кода традиционной операционной системы (обеспечивающего расширенную машину) в верхний уровень, систему CMS. Однако VM/370 и при этом останется сложной комплексной программой, потому что моделирование нескольких виртуальных 370-х машин само по себе не так просто (особенно если вы хотите сделать это достаточно эффективно).

В развитии современных операционных систем наблюдается тенденция в сторону дальнейшего переноса кода в верхние уровни и удалении при этом всего, что только возможно, из режима ядра, оставляя минимальное **микроядро**. Обычно это осуществляется перекладыванием выполнения большинства задач операционной системы на средства пользовательских процессов. Получая запрос на какую-либо операцию, например чтение блока файла, пользовательский процесс (теперь называемый **обслуживаемым процессом** или **клиентским процессом**) посылает запрос **серверному** (обслуживающему) **процессу**, который его обрабатывает и высылает назад ответ.

В модели, показанной на рис. 1.23, в задачу ядра входит только управление связью между клиентами и серверами. Благодаря разделению операционной системы на части, каждая из которых управляет всего одним элементом системы (файловой системой, процессами, терминалом или памятью), все части становятся маленькими и управляемыми. К тому же, поскольку все серверы работают как процессы в режиме пользователя, а не в режиме ядра, они не имеют прямого доступа к оборудованию. Поэтому если происходит ошибка на файловом сервере, может разрушиться служба обработки файловых запросов, но это обычно не приводит к остановке всей машины целиком.

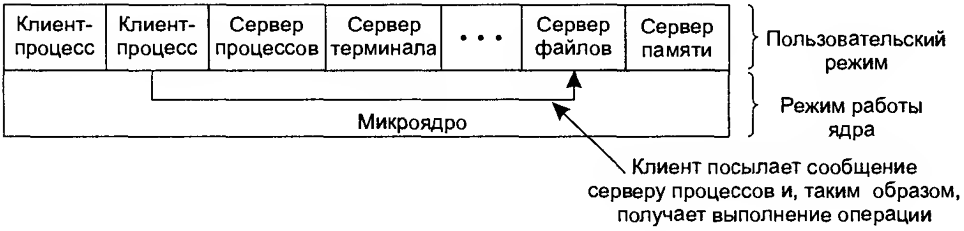


Рис. 1.23. Модель клиент-сервер

Другое преимущество модели клиент-сервер заключается в ее простой адаптации к использованию в распределенных системах (рис. 1.24). Если клиент общается с сервером, посылая ему сообщения, клиенту не нужно знать, обрабатывается ли его сообщение локально на его собственной машине или оно было послано по сети серверу на удаленной машине. С точки зрения клиента происходит одно и то же в обоих случаях: запрос был послан, и на него получен ответ.

Рассказанная выше история о ядре, управляющем передачей сообщений от клиентов к серверам и назад, не совсем реалистична. Некоторые функции операционной системы, такие как загрузка команд в регистры физических устройств ввода-вывода, трудно, если вообще возможно, выполнить из программ в пространстве пользователя. Есть два способа разрешения этой проблемы. Первый заключается в том, что некоторые критические серверные процессы (например, драйверы устройств ввода-вывода) действительно запускаются в режиме ядра, с полным доступом к аппаратуре, но при этом общаются с другими процессами при помощи обычной схемы передачи сообщений.

Второй способ состоит в том, чтобы встроить минимальный **механизм обработки информации** в ядро, но оставить принятие **политических** решений за серверами в пользовательском пространстве [202]. Например, ядро может опознавать сообщения, посланные по определенным адресам. Для ядра это означает, что нужно взять содержимое сообщения и загрузить его, скажем, в регистры ввода-вывода некоторого диска для запуска операции чтения диска. В этом примере ядро даже может не обследовать байты сообщения, если они оказались допустимы или осмысленны; оно может вслепую копировать их в регистры диска. (Очевидно, должна использоваться некоторая схема, ограничивающая круг процессов, имеющих право отправлять подобные сообщения.) Разделение между механизмом и политикой является очень важным понятием, встречающимся в операционных системах в различном контексте постоянно.

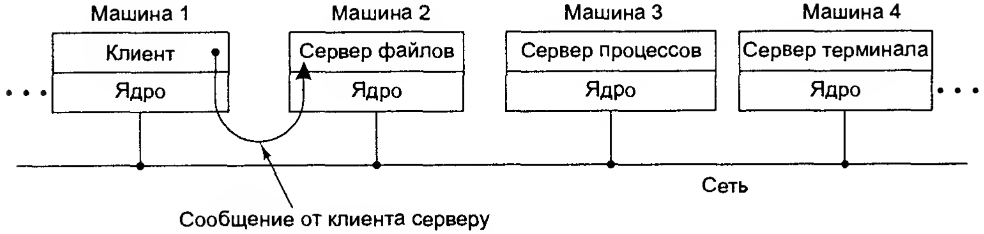


Рис. 1.24. Модель клиент-сервер в распределенной системе

Исследования в области операционных систем

Кибернетика — это быстро прогрессирующая область, поэтому, говоря о ней, крайне тяжело предсказывать будущее. Исследователи в университетах и коммерческих исследовательских лабораториях постоянно выдают новые идеи, некоторые из них не развиваются дальше, а другие закладываются в основу будущих программных продуктов и оказывают огромное влияние на производителей и пользователей. Отличить первые от вторых оказывается значительно легче задним умом, нежели в режиме реального времени. Отделить зерна от плевел трудно, поэтому часто проходит 20–30 лет между зарождением идеи и ее расцветом.

Например, когда президент Эйзенхауэр (Eisenhower) учредил в 1958 году в Министерстве обороны управление ARPA (Advanced Research Projects Agency — управление перспективных исследовательских программ), он пытался не допустить уничтожения армией флота и военно-воздушных сил из-за проблем исследовательского бюджета Пентагона. Он вовсе не пытался изобрести Интернет. Но среди прочего ARPA создала фонд для исследований некоторых университетов, посвященных тогда еще неясной идее пакетной коммутации, которая в скором времени легла в основу первой экспериментальной сети с коммутацией пакетов ARPANET. Она появилась в 1969 году. Вскоре другие финансируемые ARPA исследовательские сети присоединились к ARPANET, и так родился Интернет (Internet). Затем Интернет успешно использовался в течение 20 лет академическими исследователями. Они посылали друг другу письма по электронной почте. В начале 1990-х годов Тим Бернерс-Ли (Tim Berners-Lee) изобрел Всемирную паутину (World Wide Web) в исследовательской лаборатории CERN в Женеве, а Марк Андресен (Marc Andreesen) в университете Иллинойса написал для нее графический браузер. После этого Интернет заполнился болтающими подростками, чего явно не ожидал президент Эйзенхауэр.

Исследования в области операционных систем также привели к драматическим изменениям в практических системах. Как уже говорилось ранее, все первые коммерческие компьютерные системы были системами пакетной обработки до тех пор, пока в начале 60-х в Массачусетсском технологическом институте не изобрели интерактивные системы с разделением времени. Компьютеры были текстовыми машинами, пока в конце 60-х Даг Энгельбарт (Doug Engelbart) из Стэнфордского исследовательского института не изобрел мышь и графический интерфейс пользователя. Кто знает, что появится вслед за этим?

В этом и последующих разделах книги мы кратко опишем часть из исследований в области операционных систем, которые имели место за последние 5–10 лет, чтобы дать представление о том, что может появиться в будущем. Это введение, конечно, не является всеобъемлющим и в основном базируется на статьях, опубликованных в лучших исследовательских журналах и материалах конференций, потому что эти идеи, по крайней мере, пережили тщательный беспристрастный процесс анализа перед публикацией. Большинство научных статей, цитируемых в посвященных исследованиям разделах, были опубликованы либо ассоциацией по вычислительной технике ACM, либо компьютерным обществом IEEE, либо

USENIX. Кроме того, они доступны в Интернете для членов этих организаций, в том числе студентов. За дополнительной информацией об этих организациях и их цифровых библиотеках следует обращаться на следующие сайты:

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

На самом деле все исследователи понимают, что современные операционные системы массивны, не гибки, ненадежны, не обеспечивают защиты информации и изобилуют ошибками, причем некоторые системы в большей степени, чем остальные (*названия не сообщаются, чтобы защитить виновников*). Естественно, огромное количество исследований было посвящено построению гибкой и надежной системы. Многие исследования касаются систем микроядра. Ядра таких систем имеют минимальные размеры, поэтому есть шанс, что их смогут целиком отладить и сделать надежными. При этом такие системы гибки, потому что большая часть операционной системы работает как процессы пользовательского режима и поэтому их легко переместить или адаптировать к новым условиям, возможно, даже во время работы. Обычно в задачу микроядра входит только управление на низком уровне и передача сообщений между процессами пользователя.

Микроядра первого поколения, такие как Amoeba [324], Chorus [282], Mach [4] и V [66], доказали, что эти системы можно построить и заставить работать. Второе поколение пытается доказать, что они не только работают, но и делают это с высокой производительностью [119, 147, 208, 209, 270, 371]. Основываясь на опубликованных измерениях, можно утверждать, что эта цель достигнута.

Множество исследований в области ядра в наши дни фокусируется на конструировании расширяемых операционных систем. Это обычно системы с микроядром, в которых поддерживается возможность их расширения или переделки в некотором направлении. Такими примерами являются Fluke [121], Paramecium [137], SPIN [28] и Vino [299]. Некоторые исследователи также ищут возможности расширения существующих систем [126]. Многие из этих систем позволяют пользователям добавлять свой собственный код к ядру, но при этом встает очевидная проблема обеспечения безопасности надстроек пользователя. Среди методов разрешения этих проблем встречаются такие, как интерпретация расширений, изоляция на время выполнения загружаемого удаленного кода в ограниченную среду, так называемую «песочницу», использование языков, обеспечивающих типовую безопасность, и снабжение исполняемых программ электронной подписью [137, 306]. В [100] представлена другая точка зрения: на обеспечение безопасности расширяемых пользователями систем тратится слишком много усилий. По мнению авторов, исследователи должны вычислить, какие надстройки полезны, и затем сделать их нормальной частью ядра, не предоставляя пользователям возможностей расширять ядро на лету.

Хотя один подход к улучшению огромных, содержащих массу ошибок, ненадежных операционных систем состоит в уменьшении их размеров, существует второй, более радикальный метод — вообще устранить операционную систему. Он был взят на вооружение группой Каашойка (Kaashoek) в Массачусетсском техно-

логическом институте в их работе над экзоядром. Идея состоит в том, чтобы оставить тонкий слой программного обеспечения, работающего напрямую с аппаратурой, все действия которого заключаются в надежном распределении ресурсов аппаратуры между пользователями. Например, оно должно решать, кто и какую часть диска получает в пользование, куда должны доставляться приходящие сетевые пакеты. Все остальное оставляется на усмотрение процессов пользовательского уровня, что делает возможным построение как универсальных, так и узкоспециализированных систем [110, 108, 168].

Краткий обзор следующих глав

Итак, мы завершили введение и описали общие моменты операционных систем. Теперь пришло время разобраться в деталях. Глава 2 посвящена процессам. Здесь обсуждаются их свойства и то, как они общаются друг с другом, а также представлено несколько подробных примеров межпроцессного общения и того, как можно избежать некоторых ошибок.

В главе 3 говорится о взаимоблокировке. В этой главе мы дали некоторое представление о том, что такое взаимоблокировка, но этой теме следует посвятить отдельную главу. Кроме того, обсуждаются способы, позволяющие предупредить или избежать взаимоблокировки.

В главе 4 мы детально изучим управление памятью. Будет исследована важная часть виртуальной памяти наряду с тесно связанными с ней идеями разбиения памяти на страницы и сегменты.

Вводу-выводу посвящена глава 5. В ней будут рассмотрены концепции зависимости и независимости от устройств. В качестве примеров мы будем использовать такие важные устройства, как диски, клавиатуры и мониторы.

Затем в главе 6 мы поговорим на очень важную тему файловых систем. Ведь с файловой системой пользователь чаще всего имеет дело. Мы рассмотрим и интерфейс файловой системы, и ее реализацию.

На этом мы завершим изучение основных принципов работы однопроцессорных операционных систем. Однако мы можем рассказать еще многое, особенно о расширенных возможностях. Глава 7 посвящена изучению мультимедийных систем, имеющих ряд свойств и требований, отличающихся от традиционных операционных систем. Природа мультимедиа, кроме всего прочего, оказывает влияние на планирование и файловую систему. Есть еще одна тема для обсуждения — системы с несколькими процессорами, включая многопроцессорные системы, параллельные компьютеры и распределенные системы. Ее мы рассмотрим в главе 8.

Чрезвычайно важному вопросу безопасности операционных систем посвящена глава 9. Среди прочих вопросов в этой главе обсуждаются различные виды потенциальной опасности (например, вирусы и черви), механизмы защиты и модели охраны.

Затем мы изучим некоторые существующие операционные системы: UNIX (глава 10) и Windows 2000 (глава 11). Книга заканчивается главой 12, содержащей некоторые соображения по поводу проектирования операционных систем.

Единицы измерения

Чтобы избежать путаницы в будущем, необходимо особо отметить то, что в этой книге, как книге по науке о компьютерах вообще, используются единицы метрической системы вместо традиционных английских единиц измерения (система фарлонг-стоун-дюжина). Основные метрические префиксы перечислены в табл. 1.4. Обычно префиксы сокращаются до первых букв, причем, если единица измерения больше 1, используются заглавные буквы. Например, база данных размером в 10 Тбайт занимает около 10^{12} байт на диске, а часы с интервалом в 100 пс будут тикать каждую 10^{-12} с. Так как приставки милли- и микро- начинаются с буквы «м», нужно было выбрать для них разные сокращения. Обычно для милли- используется «м», а для микро- — сокращение «мк».

Таблица 1.4. Основные метрические префиксы

Показатель	Явный вид	Префикс	Показатель	Явный вид	Префикс
10^{-3}	0,001	милли	10^{3}	1000	Кило
10^{-6}	0,000001	микро	10^{6}	1 000 000	Мега
10^{-9}	0,000000001	нано	10^{9}	1 000 000 000	Гига
10^{-12}	0,000000000001	пико	10^{12}	1 000 000 000 000	Тера
10^{-15}	0,000000000000001	фемто	10^{15}	1 000 000 000 000 000	Пета
10^{-18}	0,000000000000000001	атто	10^{18}	1 000 000 000 000 000 000	Екза
10^{-21}	0,000000000000000000001	зепто	10^{21}	1 000 000 000 000 000 000 000	Зетта
10^{-24}	0,000000000000000000000001	йокто	10^{24}	1 000 000 000 000 000 000 000 000	Йотта

При измерении размеров памяти в компьютерной промышленности принято использовать единицы, значения которых несколько отличаются от общепринятых. Здесь Кило обозначает 2^{10} (1024), то есть немного больше, чем 10^3 (1000), но память всегда измеряется в степенях двойки. Таким образом, 1 Кбайт содержит 1024 байта, а не 1000 байт. Точно так же 1 Мбайт содержит 2^{20} (1 048 576) байт, а 1 Гбайт памяти равен 2^{30} байт (1 073 741 824). Однако коммуникационная линия, передающая 1 Кбит/с, на самом деле передает 1000 бит/с, а 10-мегабитная локальная сеть работает со скоростью 10 000 000 бит/с, потому что эти скорости не являются степенью двойки. К сожалению, многие люди смешивают эти две системы, особенно когда говорят о размерах диска. Чтобы избежать неясности, мы будем использовать символы Кбайт, Мбайт и Гбайт для 2^{10}, 2^{20} и 2^{30} соответственно, а Кбит/с, Мбит/с и Гбит/с для 10^3, 10^6 и 10^9 бит/с соответственно.

Резюме

Операционную систему можно рассматривать с двух точек зрения: как менеджер ресурсов и как расширенную машину. Как менеджер ресурсов операционная система рационально управляет различными частями системы. С точки зрения расширенной машины, работа операционной системы состоит в предоставлении пользователям виртуальной машины, более удобной, чем настоящий компьютер.

Операционные системы имеют достаточно долгую историю развития, которая начинается с тех дней, когда операционные системы заменили оператора, и продолжается до современных многозадачных систем. Большое значение имеют ранние системы пакетной обработки, многозадачные системы и системы для персональных компьютеров.

Поскольку операционные системы тесно взаимодействуют с оборудованием, некоторые знания об аппаратуре могут оказаться очень полезны для понимания работы операционной системы. Компьютеры состоят из процессоров, памяти и устройств ввода-вывода. Все эти части соединены шинами.

Основными понятиями, на которых построена операционная система, являются процессы, управление памятью, управление вводом-выводом, файловая система и безопасность. Каждое из них будет разобрано в соответствующей главе.

Сердцем любой операционной системы является набор системных вызовов, которые она может обработать. Они говорят о том, что реально делает операционная система. Мы рассмотрели четыре группы системных вызовов для UNIX. Первая группа работает с созданием и завершением процессов. Вторая группа предназначена для чтения и записи файлов. Третья нужна для управления каталогами. Четвертая включила в себя различные другие вызовы.

Операционная система может быть структурирована несколькими способами. Наиболее общими выделяемыми при структурировании понятиями являются: монолитные системы, иерархия слоев, система виртуальных машин, экзоядро или использование модели клиент-сервер.

Вопросы

1. Каковы две главные функции операционной системы?
2. Что такое многозадачность?
3. Что такое подкачка данных (spooling)? Как вы считаете, будут ли передовые персональные компьютеры иметь в будущем подкачку данных в качестве стандартного элемента?
4. На ранних компьютерах чтение или запись каждого байта данных управлялось напрямую центральным процессором (то есть тогда не было прямого доступа к памяти — DMA). Какой смысл имеет это понятие для многозадачности?
5. Почему системы с разделением времени не были широко распространены на компьютерах второго поколения?
6. Идея семейства компьютеров родилась в 60-е годы вместе с появлением мэйнфреймов серии IBM System/360. Сейчас эта идея считается устаревшей или все еще имеет силу?
7. Одной из причин, по которой на начальной стадии очень медленно распространялся графический интерфейс (GUI), была высокая стоимость необходимого оборудования. Сколько нужно оперативной видеопамяти для поддержки монохромного текстового экрана размером 25×80 строк? И сколько

необходимо для 24-битового цветного отображения на экране размером 1024 × 768 пикселов? Какова была стоимость такой видеопамяти в ценах 1980 года (5 долларов за килобайт)? Сколько она стоит сейчас?

8. Какая из следующих команд должна быть разрешена только в режиме ядра:
   а) отключение всех прерываний;
   б) чтение счетчика даты/времени;
   в) изменения счетчика даты/времени;
   г) изменение схемы распределения памяти.
9. Перечислите основные различия между операционной системой для персонального компьютера и для мэйнфрейма.
10. В компьютере есть конвейер, состоящий из четырех ступеней. Каждая из них выполняет свою работу за одно и то же время, а именно, за 1 нс. Сколько команд в секунду может выполнить машина?
11. Внимательный рецензент замечает несущественную орфографическую ошибку в рукописи книги по операционным системам, которую собираются печатать. Книга состоит примерно из 700 страниц, на каждой 50 строк по 80 знаков. Сколько времени займет сканирование текста, если копия будет целиком размещаться на каждом из уровней памяти, представленных на рис. 1.7? Для методов внутреннего хранения считаем, что время доступа дано на знак, для диска дано время предположительно на блок из 1024 знаков и для ленты полагаем, что данное время означает последовательный доступ к данным, с той же самой скоростью, что и для диска.
12. На рис. 1.9 блок управления памятью MMU сравнивает входящий (виртуальный) адрес с предельным регистром, вызывая ошибку, если адрес слишком большой. Альтернативным решением будет сначала сложить виртуальный адрес с индексным регистром и затем сравнить результат (физический адрес) с предельным регистром. Эквивалентны ли эти два метода логически? Эквивалентны ли эти они с точки зрения производительности?
13. Когда пользовательская программа выдает системный вызов, чтобы прочитать или записать файл с диска, она сопровождает его информацией о том, какой файл ей нужен, указателем на буфер данных и количеством байт. Затем управление передается операционной системе, вызывающей подходящий драйвер. Предположим, что драйвер работает с диском и прекращает свое действие до тех пор, пока не произойдет прерывание. В случае чтения данных с диска, очевидно, вызывающая программа должна быть заблокирована (потому что для нее нет данных). Что происходит при записи данных на диск? Нужно ли блокировать вызывающую программу для ожидания завершения передачи данных на диск?
14. В чем заключается разница между эмулированным и аппаратным прерываниями?
15. Компьютер использует схему настройки адресов программы, показанную на рис. 1.9, *а*. Программа длиной в 10 000 байт загружена по адресу 40 000. Какие значения примут базовый и предельный регистры соответственно схеме, описанной в тексте?

16. Почему в системах разделения времени необходима таблица процессов? Нужна ли она также в системах на персональных компьютерах, где существует только один процесс, и этот процесс завладевает всей машиной до тех пор, пока не завершится?
17. Есть ли какая-нибудь причина, по которой вам может понадобиться встроить файловую систему в непустой каталог? Если да, то что это за причина?
18. Для каждого из следующих системных вызовов укажите условие, при котором возникнет ошибка: fork, exec и unlink.
19. Может ли вызов

    ```
    count = write(fd, buffer, nbytes);
    ```

    вернуть в переменной *count* величину, отличную от *nbytes*? Если да, то почему?
20. Файл, дескриптором которого является *fd*, содержит следующую последовательность байтов: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Выполняется следующий системный вызов:

    ```
    lseek(fd, 3, SEEK_SET);
    read(fd, &buffer, 4);
    ```

    где вызов lseek ищет байт 3 в файле. Что будет содержать буфер после завершения операции чтения?
21. В чем заключается существенная разница между блоковым специальным файлом и символьным специальным файлом?
22. В примере, показанном на рис. 1.17, библиотечная процедура имеет название *read*, и сам системный вызов называется read. Существенно ли то, что они имеют одинаковые имена? Если нет, то которое важнее?
23. Модель клиент-сервер популярна в распределенных системах. Можно ли ее также использовать в однокомпьютерных системах?
24. Для программиста системный вызов выглядит так же, как и любой другой вызов библиотечной процедуры. Должен ли программист знать, обращение к каким библиотечным процедурам приводит в результате к системным вызовам? При каких условиях и почему?
25. В табл. 1.2 показано, что ряд системных вызовов UNIX не имеют эквивалентов в Win32 API. Каковы последствия для программиста перевода программы UNIX для запуска под Windows (для каждого из вызовов, помеченных, как не имеющие эквивалента)?
26. Несколько задач для тренировки перевода одних единиц в другие:
    а) сколько длится микрогод в секундах?
    б) микрометры часто называют микронами. Какова длина гигамикрона?
    в) сколько байт в памяти размером в 1 Тбайт?
    г) масса земли составляет 6000 йоттаграмм. Какова она в килограммах?
27. Напишите оболочку, похожую на представленную в листинге 1.1, но содержащую все необходимое для того, чтобы она работала, так что вы смогли бы ее протестировать. Вы можете также добавить некоторые элементы, например перенаправление ввода и вывода, каналы и фоновые задания.

28. Если у вас есть персональная система из семейства UNIX (Linux, MINIX, Free BSD и т. д.), которую вам не страшно «повесить» и перезагрузиться, напишите сценарий для оболочки, который пытается создать неограниченное число дочерних процессов, запустите его и посмотрите, что произойдет. Перед запуском эксперимента наберите команду sync, чтобы сбросить на диск содержимое буферов файловой системы во избежание крушения файловой системы. *Замечание:* не пытайтесь сделать это на разделенной системе без предварительного получения разрешения от системного администратора. Последствия этой операции проявятся немедленно, поэтому вас, скорее всего, поймают и применят к вам соответствующие санкции.
29. Исследуйте и попытайтесь интерпретировать содержимое каталогов в системах семейства UNIX или Windows с помощью программ *od* системы UNIX или *DEBUG* в MS-DOS. *Совет*: то, как вы это сделаете, зависит от того, что позволяет операционная система. Есть одна уловка, которая может сработать: создайте каталог на гибком диске в одной операционной системе, а затем считайте необработанные данные диска, используя другую операционную систему, которая позволяет подобный доступ.

Глава 2
Процессы и потоки

Теперь мы детально рассмотрим устройство и работу операционных систем. Основным понятием, связанным с операционными системами, является *процесс* — абстрактное понятие, описывающее работу программы. Все остальное базируется на этом понятии, поэтому представляется крайне важным, чтобы разработчики операционных систем (а также студенты) получили полное представление о концепции процесса как можно раньше.

Процессы

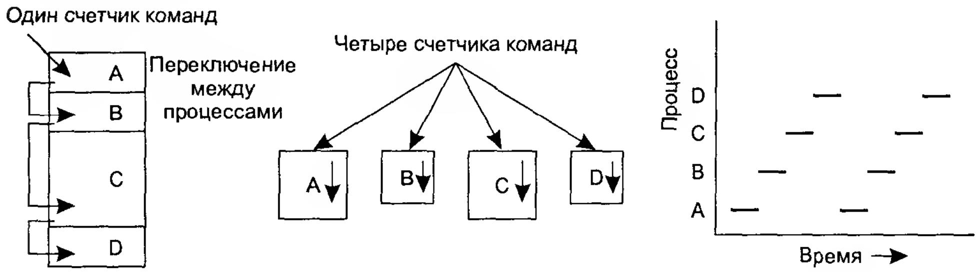
Все современные компьютеры могут делать одновременно несколько дел. Например, одновременно с запущенной пользователем программой может выполняться чтение с диска и вывод текста на экран монитора или на принтер. В многозадачной системе процессор переключается между программами, предоставляя каждой от десятков до сотен миллисекунд. При этом в каждый конкретный момент времени процессор занят только одной программой, но за секунду он успевает поработать с несколькими программами, создавая у пользователей иллюзию параллельной работы со всеми программами. Иногда в этом контексте говорят о **псевдопараллелизме**, в отличие от настоящего параллелизма в **многопроцессорных** системах (в которых установлено два и более процессора, разделяющих между собой общую физическую память). Следить за работой параллельно идущих процессов достаточно трудно, поэтому со временем разработчики операционных систем разработали концептуальную модель последовательных процессов, упрощающую эту работу. Темой данной главы будет содержание и применение этой модели, а также некоторые результаты ее применения.

Модель процесса

В этой модели все функционирующее на компьютере программное обеспечение, иногда включая собственно операционную систему, организовано в виде набора **последовательных процессов**, или, для краткости, просто **процессов**. Процессом является выполняемая программа, включая текущие значения счетчика команд, регистров и переменных. С позиций данной абстрактной модели, у каждого процесса есть собственный виртуальный центральный процессор. На самом деле, разумеется, реальный процессор переключается с процесса на процесс, но для лучшего

понимания системы значительно проще рассматривать набор процессов, идущих параллельно (псевдопараллельно), чем пытаться представить себе процессор, переключающийся от программы к программе. Как мы уже знаем из первой главы, это переключение и называется **многозадачностью** или **мультипрограммированием**.

На рис. 2.1, *а* представлена схема компьютера, работающего с четырьмя программами. На рис. 2.1, *б* представлены четыре процесса, каждый со своей управляющей логикой (то есть логическим счетчиком команд), идущие независимо друг от друга. Разумеется, на самом деле существует только один физический счетчик команд, в который загружается логический счетчик команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в логическом счетчике команд процесса в памяти. На рис. 2.1, *в* видно, что за достаточно большой промежуток времени изменилось состояние всех четырех процессов, но в каждый конкретный момент в действительности работает только один процесс.



а *б* *в*

Рис. 2.1. Четыре программы в многозадачном режиме (*а*); принципиальная модель четырех независимых последовательных процессов (*б*); в каждый момент времени активна только одна программа (*в*)

Поскольку процессор переключается между программами, скорость, с которой процессор производит свои вычисления, будет непостоянной и, возможно, даже будет отличной при каждом новом запуске процесса. Поэтому не следует программировать процессы, исходя из каких-либо жестко заданных временных предположений. Представьте себе, например, процесс ввода-вывода, запускающий накопитель на магнитной ленте для восстановления заархивированных файлов. Процесс выполняет холостой цикл задержки 10 000 раз, чтобы дать время накопителю разогнаться, а затем дает команду считать первый сектор. Если во время холостого цикла процессор решит переключиться на другую задачу, может случиться так, что работающий с магнитофоном процесс запустится снова уже после того, как считывающая головка пройдет первую запись. Если у процесса есть критические временные рамки такого рода, то есть отдельные события должны укладываться в заданное количество миллисекунд, необходимы специальные меры, чтобы удостовериться в завершенности события. Однако обычно многозадачный режим процессора, а также относительные скорости различных процессов не влияют на работу большинства процессов.

Различие между процессом и программой трудноуловимо и тем не менее имеет принципиальное значение. Воспользуемся следующей аналогией: представьте

себе программиста, разбирающегося в кулинарии и пекущего торт на день рождения своей дочери. В его распоряжении есть рецепт торта, кухня, оборудованная всем необходимым, и ингредиенты для торта: мука, яйца, сахар, ванилин и т. п. Согласно этой аналогии, рецепт — это программа (то есть алгоритм, записанный в заданном виде), программист исполняет роль процессора, а ингредиенты торта являются входными данными. Процессом является следующая последовательность действий: программист читает рецепт, смешивает продукты и печет торт.

Теперь представьте, что на кухню прибегает плачущий сын программиста и кричит, что его ужалила пчела. Программист отмечает, на чем он остановился (сохраняет текущее состояние процесса), находит справочник по оказанию первой помощи и действует в соответствии с инструкцией. Таким образом, наш процессор переключился с одного процесса (выпечка торта) на другой, с большим приоритетом (оказание первой помощи), и у каждого процесса есть своя программа (рецепт торта и справочник по оказанию первой помощи). После проведения всех необходимых процедур по борьбе с укусом пчелы программист возвращается к торту, продолжая с той операции, на которой он прервался.

Мы привели эту аналогию с целью показать, что процесс — это активность некоторого рода. У него есть программа, входные и выходные данные, а также состояние. Один процессор может переключаться между различными процессами, используя некий алгоритм планирования для определения момента переключения от одного процесса к другому.

Создание процесса

Операционной системе необходим способ, позволяющий удостовериться в наличии всех необходимых процессов. В простейших системах, а также системах, разработанных для выполнения одного-единственного приложения (например, контроллер микроволновой печи), можно реализовать такую ситуацию, в которой все процессы, которые когда-либо могут понадобиться, присутствуют в системе при ее загрузке. В универсальных системах необходим способ создания и прерывания процессов по мере необходимости. В этом разделе мы рассмотрим некоторые из возможных способов решения этой проблемы. Ниже перечислены четыре основных события, приводящие к созданию процессов.

1. Инициализация системы.
2. Выполнение изданного работающим процессом системного запроса на создание процесса.
3. Запрос пользователя на создание процесса.
4. Инициирование пакетного задания.

Обычно при загрузке операционной системы создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, то есть обеспечивающими взаимодействие с пользователем и выполняющими заданную работу. Остальные процессы являются фоновыми, они не связаны с конкретными пользователями, но выполняют особые функции. Например, один фоновый процесс может быть предназначен для обработки приходящей на компьютер почты,

активизируясь только по мере появления писем. Другой фоновый процесс может обрабатывать запросы к web-страницам, расположенным на компьютере, и активизироваться для обслуживания полученного запроса. Фоновые процессы, связанные с электронной почтой, web-страницами, новостями, выводом на печать и т. п., называются **демонами.** В больших системах насчитываются десятки демонов. В UNIX для вывода списка запущенных процессов используется программа ps. В Windows 95/98/Me достаточно нажать CTRL-ALT-DEL, а в Windows 2000 можно воспользоваться диспетчером задач, вызываемым этой же комбинацией трех клавиш.

Процессы могут создаваться не только в момент загрузки системы, но и позже. Например, новый процесс (или несколько) может быть создан по просьбе текущего процесса. Создание новых процессов особенно полезно в тех случаях, когда выполняемую задачу проще всего сформировать как набор связанных, но тем не менее независимых взаимодействующих процессов. Если необходимо организовать выборку большого количества данных из сети для дальнейшей обработки, удобно создать один процесс для выборки данных и размещения их в совместно используемом буфере, в то время как второй процесс будет считывать данные из буфера и обрабатывать их. Эта схема даже ускорит обработку данных, если каждый процесс запустить на отдельном процессоре в случае многопроцессорной системы.

В интерактивных системах пользователь может запустить программу, набрав на клавиатуре команду или дважды щелкнув на значке программы. В обоих случаях результатом будет создание нового процесса и запуск в нем программы. Когда на UNIX работает X Windows, новый процесс получает то окно, в котором был запущен. В Microsoft Windows процесс не имеет собственного окна при запуске, но он может (и должен) создать одно или несколько окон. В обеих системах пользователь может одновременно открыть несколько окон, каждому иэ которых соответствует свой процесс. Пользователь может переключаться между окнами с помощью мыши и взаимодействовать с процессом, например, вводя данные по мере необходимости.

Последнее событие, приводящее к созданию нового процесса, связано с системами пакетной обработки на больших компьютерах. Пользователи посылают пакетное задание (возможно, с использованием удаленного доступа), а операционная система создает новый процесс и запускает следующее задание из очереди в тот момент, когда освобождаются необходимые ресурсы.

С технической точки зрения во всех перечисленных случаях новый процесс формируется одинаково: текущий процесс выполняет системный запрос на создание нового процесса. В роли текущего процесса может выступать процесс, запущенный пользователем, системный процесс, инициированный клавиатурой или мышью, а также процесс, управляющий пакетами. В любом случае этот процесс всего лишь выполняет системный запрос и создает новый процесс. Системный запрос заставляет операционную систему создать новый процесс, а также прямо или косвенно содержит информацию о программе, которую нужно запустить в этом процессе.

В UNIX существует только один системный запрос, направленный на создание нового процесса: fork (*ветвление* или *вилка*). Этот запрос создает дубликат вызываемого процесса. После выполнения запроса fork двум процессам — родительскому и дочернему — соответствуют одинаковые образы памяти, строки окружения и одни и те же открытые файлы. Обычно дочерний процесс выполняет системный

вызов execve (или похожий) для изменения своего образа памяти и запуска новой программы. Так, когда пользователь набирает на клавиатуре команду *sort*, оболочка создает путем ветвления дочерний процесс, который и выполняет программу *sort*. Смысл этого двухступенчатого процесса заключается в том, что дочерний процесс успевает обработать описания файлов после fork, но до execve, чтобы выполнить перенаправление стандартных устройств ввода и вывода и потока сообщений об ошибках.

В Windows же вызов всего одной функции CreateProcess интерфейса Win32 управляет и созданием процесса, и запуском в нем нужной программы. У этой функции 10 параметров: программа, которую нужно запустить, параметры командной строки этой программы, различные атрибуты защиты, биты, управляющие наследованием открытых файлов, приоритеты, спецификация окна, которое следует открыть для процесса, и указатель на структуру, в которой информация о созданном процессе возвращается вызывающей программе. Кроме CreateProcess в Win32 есть около 100 функций для управления процессами и их синхронизации.

И в UNIX, и в Windows после создания нового процесса родительский и дочерний процессы имеют собственные различные адресные пространства. При изменении любым процессом слова в адресном пространстве это изменение незаметно для других процессов. В UNIX начальное адресное пространство дочернего процесса является *копией* родительского, но сами адресные пространства различны, и перезаписываемая память совместно не используется (некоторые приложения UNIX совместно используют текст программы, поскольку его нельзя модифицировать). В то же время созданный процесс может использовать совместно с родительским процессом некоторые другие ресурсы, например открытые файлы. В Windows адресные пространства родительского и дочернего процессов отличаются с самого начала.

Завершение процесса

После того как процесс создан, он начинает выполнять свою работу. Но ничто не длится вечно, даже процесс — рано или поздно он завершится, чаще всего благодаря одному из следующих событий:

1. Обычный выход (преднамеренно).
2. Выход по ошибке (преднамеренно).
3. Выход по неисправимой ошибке (непреднамеренно).
4. Уничтожение другим процессом (непреднамеренно).

В основном процессы завершаются по мере выполнения своей работы. После окончания компиляции программы компилятор выполняет системный запрос, чтобы сообщить операционной системе об окончании работы. В UNIX этот системный запрос — exit, а в Windows — ExitProcess. Программы, рассчитанные на работу с экраном, также поддерживают преднамеренное завершение. В текстовых редакторах, браузерах и других программах такого типа обычно есть кнопка или пункт меню, щелкнув на котором можно удалить все временные файлы, открытые процессом, и затем завершить процесс.

Второй причиной завершения процесса может стать неустранимая ошибка. Например, если пользователь набрал на клавиатуре команду

```
cc foo.c
```

для компиляции программы *foo.c*, а соответствующего файла не существует, компилятор просто закончит работу. Интерактивные процессы, рассчитанные на работу с экраном, обычно не завершают работу при получении неверных параметров, вместо этого выводя на экран диалоговое окно и прося пользователя ввести правильные параметры.

Третьей причиной завершения процесса является ошибка, вызванная самим процессом, чаще всего связанная с ошибкой в программе. В качестве примера можно привести выполнение недопустимой команды, обращение к несуществующей области памяти и деление на ноль. В некоторых системах (например, в UNIX) процесс может информировать операционную систему о том, что он сам обработает некоторые ошибки, и в этом случае процессу посылается сигнал (процесс прерывается, а не завершается) при появлении ошибки.

Четвертой причиной завершения процесса может служить выполнение другим процессом системного запроса на уничтожение процесса. В UNIX такой системный запрос — `kill`, а соответствующая функция Win32 — `TerminateProcess`. В обоих случаях «киллер» должен обладать соответствующими полномочиями по отношению к «убиваемому» процессу. В некоторых системах при завершении процесса (преднамеренно или нет) все процессы, созданные процессом, также завершаются. Впрочем, это не относится ни к UNIX, ни к Windows.

Иерархия процессов

В некоторых системах родительский и дочерний процессы остаются связанными между собой определенным образом. Дочерний процесс также может, в свою очередь, создавать процессы, формируя *иерархию* процессов. Следует отметить, что в отличие от животного мира у процесса может быть лишь один родитель и сколько угодно «детей».

В UNIX процесс, все его «дети» и дальнейшие потомки образуют *группу процессов*. Сигнал, посылаемый пользователем с клавиатуры, доставляется всем членам группы, взаимодействующим с клавиатурой в данный момент (обычно это все активные процессы, созданные в текущем окне). Каждый из процессов может перехватить сигнал, игнорировать его или выполнить другое действие, предусмотренное по умолчанию.

Рассмотрим в качестве еще одного примера иерархии процессов инициализацию UNIX при запуске. В образе загрузки присутствует специальный процесс *init*. При запуске этот процесс считывает файл, в котором находится информация о количестве терминалов. Затем процесс разветвляется таким образом, чтобы каждому терминалу соответствовал один процесс. Процессы ждут, пока какой-нибудь пользователь не войдет в систему. Если пароль правильный, процесс входа в систему запускает оболочку для обработки команд пользователя, которые, в свою очередь, могут запускать процессы. Таким образом, все процессы в системе принадлежат к единому дереву, начинающемуся с процесса *init*. Напротив, в Windows не суще-

ствует понятия иерархии процессов, и все процессы равноправны. Единственное, в чем проявляется что-то вроде иерархии процессов — создание процесса, в котором родительский процесс получает специальный маркер (так называемый **дескриптор**), позволяющий контролировать дочерний процесс. Но маркер можно передать другому процессу, нарушая иерархию. В UNIX это невозможно.

Состояния процессов

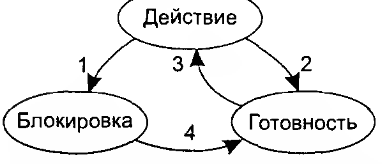
Несмотря на то что процесс является независимым объектом, со своим счетчиком команд и внутренним состоянием, существует необходимость взаимодействия с другими процессами. Например, выходные данные одного процесса могут служить входными данными для другого процесса. В команде оболочки

```
cat chapter1 chapter2 chapter3 | grep tree
```

первый процесс, исполняющий файл cat, объединяет и выводит три файла. Второй процесс, исполняющий файл grep, отбирает все строки, содержащие слово «tree». В зависимости от относительных скоростей процессов (скорости зависят от относительной сложности программ и процессорного времени, предоставляемого каждому процессу), может получиться, что grep уже готов к запуску, но входных данных для этого процесса еще нет. В этом случае процесс блокируется до поступления входных данных.

Процесс блокируется, поскольку с точки зрения логики он не может продолжать свою работу (обычно это связано с отсутствием входных данных, ожидаемых процессом). Также возможна ситуация, когда процесс, готовый и способный работать, останавливается, поскольку операционная система решила предоставить на время процессор другому процессу. Эти ситуации являются принципиально разными. В первом случае приостановка выполнения является внутренней проблемой (поскольку невозможно обработать командную строку пользователя до того, как она была введена). Во втором случае проблема является технической (нехватка процессоров для каждого процесса). На рис. 2.2 представлена диаграмма состояний, показывающая три возможных состояния процесса:

1. Работающий (в этот конкретный момент использующий процессор).
2. Готовый к работе (процесс временно приостановлен, чтобы позволить выполняться другому процессу).
3. Заблокированный (процесс не может быть запущен прежде, чем произойдет некое внешнее событие).



1. Процесс блокируется, ожидая входных данных
2. Планировщик выбирает другой процесс
3. Планировщик выбирает этот процесс
4. Доступны входные данные

Рис. 2.2. Процесс может находиться в рабочем, готовом и заблокированном состоянии. Стрелками показаны возможные переходы между состояниями

С точки зрения логики первые два состояния одинаковы. В обоих случаях процесс может быть запущен, только во втором случае недоступен процессор. Третье состояние отличается тем, что запустить процесс невозможно, независимо от загруженности процессора.

Как показано на рис. 2.2, между этими тремя состояниями возможны четыре перехода. Переход *1* происходит, когда процесс обнаруживает, что продолжение работы невозможно. В некоторых системах процесс должен выполнить системный запрос, например `block` или `pause`, чтобы оказаться в заблокированном состоянии. В других системах, как в UNIX, процесс автоматически блокируется, если при считывании из канала или специального файла (предположим, терминала) входные данные не были обнаружены.

Переходы *2* и *3* вызываются частью операционной системы, называемой планировщиком процессов, так что сами процессы даже не знают о существовании этих переходов. Переход *2* происходит, если планировщик решил, что пора предоставить процессор следующему процессу. Переход *3* происходит, когда все остальные процессы уже исчерпали свое процессорное время, и процессор снова возвращается к первому процессу. Вопрос планирования (когда следует запустить очередной процесс и на какое время) сам по себе достаточно важен, и мы вернемся к нему позже в этой главе. Было разработано множество алгоритмов с целью сбалансировать требования эффективности для системы в целом и для каждого процесса в отдельности. Мы также рассмотрим некоторые из них ниже в этой главе.

Переход *4* происходит с появлением внешнего события, ожидавшегося процессом (например, прибытие входных данных). Если в этот момент не запущен какой-либо другой процесс, то срабатывает переход *3*, и процесс запускается. В противном случае процессу придется некоторое время находиться в состоянии готовности, пока не освободится процессор.

Модель процессов упрощает представление о внутреннем поведении системы. Некоторые процессы запускают программы, выполняющие команды, введенные с клавиатуры пользователем. Другие процессы являются частью системы и обрабатывают такие задачи, как выполнение запросов файловой службы, управление запуском диска или магнитного накопителя. В случае дискового прерывания система останавливает текущий процесс и запускает дисковый процесс, который был заблокирован в ожидании этого прерывания. Вместо прерываний мы можем представлять себе дисковые процессы, процессы пользователя, терминала и т. п., блокирующиеся на время ожидания событий. Когда событие произошло (информация прочитана с диска или клавиатуры), блокировка снимается и процесс может быть запущен.

Рассмотренный подход описывается моделью, представленной на рис. 2.3. Нижний уровень операционной системы — это планировщик, на верхних уровнях расположено множество процессов. Вся обработка прерываний и детали, связанные с остановкой и запуском процессов, спрятаны в том, что мы назвали планировщиком, являющимся, по сути, совсем небольшой программой. Вся остальная часть операционной системы удобно структурирована в виде набора процессов. Очень немногие существующие системы структурированы столь удобно.



Рис. 2.3. Нижний уровень операционной системы отвечает за прерывания и планирование. Выше расположены последовательные процессы

Реализация процессов

Для реализации модели процессов операционная система содержит таблицу (массив структур), называемую **таблицей процессов**, с одним элементом для каждого процесса. (Некоторые авторы называют эти элементы **блоками управления процессом**.) Элемент таблицы содержит информацию о состоянии процесса, счетчике команд, указателе стека, распределении памяти, состоянии открытых файлов, об использовании и распределении ресурсов, а также всю остальную информацию, которую необходимо сохранять при переключении в состояние *готовности* или *блокировки* для последующего запуска — как если бы процесс не останавливался.

В табл. 2.1 представлены некоторые наиболее важные поля типичной системы. Поля в первой колонке относятся к управлению процессом. Остальные колонки описывают управление памятью и файлами. Необходимо отметить, что от конкретной системы очень сильно зависит, какие именно поля будут в таблице процессов, но табл. 2.1 дает общее представление о необходимой информации.

Теперь, после знакомства с таблицей процессов, можно сказать еще несколько слов о том, как поддерживается иллюзия нескольких последовательных процессов на машине с одним процессором и несколькими устройствами ввода-вывода. С каждым классом устройств ввода-вывода (гибкий диск, жесткий диск, таймер, терминал) связана область памяти (обычно расположенная в нижних адресах), называемая **вектором прерываний**. Вектор прерываний содержит адрес процедуры обработки прерываний. Представьте, что в момент прерывания диска работал пользовательский процесс 3. Содержимое счетчика команд процесса, слово состояния программы и, возможно, один или несколько регистров записываются в (текущий) стек аппаратными средствами прерывания. Затем происходит переход по адресу, указанному в векторе прерывания диска. Вот и все, что делают аппаратные средства прерывания. С этого момента вся остальная обработка прерывания производится программным обеспечением, например процедурой обработки прерываний.

Все прерывания начинаются с сохранения регистров, часто в блоке управления текущим процессом в таблице процессов. Затем информация, помещенная в стек прерыванием, удаляется, и указатель стека переставляется на временный стек, используемый программой обработки процесса. Такие действия, как сохранение регистров и установка указателя стека, невозможно даже выразить на языке высокого уровня (например, на С). Поэтому они выполняются небольшой программой на ассемблере, обычно одинаковой для всех прерываний, поскольку процедура сохранения регистров не зависит от причины возникновения прерывания.

Таблица 2.1. Некоторые поля типичного элемента таблицы процессов

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на текстовый сегмент	Корневой каталог
Счетчик команд	Указатель на сегмент данных	Рабочий каталог
Слово состояния программы	Указатель на сегмент стека	Дескрипторы файла
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время начала процесса		
Использованное процессорное время		
Процессорное время дочернего процесса		
Время следующего аварийного сигнала		

По завершении своей работы эта программа вызывает процедуру на языке C, которая выполняет все остальные действия, связанные с конкретным прерыванием. (Мы предполагаем, что операционная система написана на C, что является стандартным решением для всех существующих операционных систем.) Когда процедура завершает свою работу (в результате чего, возможно, некоторые процессы переходят в состояние готовности), вызывается планировщик для выбора следующего процесса. После этого управление возвращается к программе на ассемблере, загружающей регистры и карту памяти для текущего процесса и запускающей его. Управление прерыванием и работа планировщика представлены в табл. 2.2. Следует отметить, что отдельные детали могут несколько варьироваться от системы к системе.

Таблица 2.2. Схема обработки прерывания нижним уровнем операционной системы

1. Аппаратное обеспечение сохраняет в стеке счетчик команд и т. п.
2. Аппаратное обеспечение загружает новый счетчик команд из вектора прерываний
3. Процедура на ассемблере сохраняет регистры
4. Процедура на ассемблере устанавливает новый стек
5. Запускается программа обработки прерываний на C. (Она обычно считывает и буферизирует входные данные)
6. Планировщик выбирает следующий процесс
7. Программа на C передает управление процедуре на ассемблере
8. Процедура на ассемблере запускает новый процесс

Потоки

В обычных операционных системах каждому процессу соответствует адресное пространство и одиночный управляющий поток. Фактически это и определяет процесс. Тем не менее часто встречаются ситуации, в которых предпочтительно

иметь несколько квазипараллельных управляющих потоков в одном адресном пространстве, как если бы они были различными процессами (однако разделяющим одно адресное пространство). В следующих разделах мы рассмотрим такие ситуации.

Модель потока

Модель процесса, которую мы рассматривали, базируется на двух независимых концепциях: группировании ресурсов и выполнении программы. Иногда полезно их разделять, и тут появляется понятие потока.

С одной стороны, процесс можно рассматривать как способ объединения родственных ресурсов в одну группу. У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы. Ресурсами являются открытые файлы, дочерние процессы, необработанные аварийные сообщения, обработчики сигналов, учетная информация и многое другое. Гораздо проще управлять ресурсами, объединив их в форме процесса.

С другой стороны, процесс можно рассматривать как поток исполняемых команд или просто **поток**. У потока есть счетчик команд, отслеживающий порядок выполнения действий. У него есть регистры, в которых хранятся текущие переменные. У него есть стек, содержащий протокол выполнения процесса, где на каждую процедуру, вызванную, но еще не вернувшуюся, отведен отдельный фрейм. Хотя поток должен исполняться внутри процесса, следует различать концепции потока и процесса. Процессы используются для группирования ресурсов, а потоки являются объектами, поочередно исполняющимися на центральном процессоре.

Концепция потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких программ, в достаточной степени независимых. Несколько потоков, работающих параллельно в одном процессе, аналогичны нескольким процессам, идущим параллельно на одном компьютере. В первом случае потоки разделяют адресное пространство, открытые файлы и другие ресурсы. Во втором случае процессы совместно пользуются физической памятью, дисками, принтерами и другими ресурсами. Потоки обладают некоторыми свойствами процессов, поэтому их иногда называют **упрощенными процессами**. Термин **многопоточность** также используется для описания использования нескольких потоков в одном процессе.

На рис. 2.4, *а* представлены три обычных процесса, у каждого из которых есть собственное адресное пространство и одиночный поток управления. На рис. 2.4, *б* представлен один процесс с тремя потоками управления. В обоих случаях мы имеем три потока, но на рис. 2.4, *а* каждый из них имеет собственное адресное пространство, а на рис. 2.4, *б* потоки разделяют единое адресное пространство.

При запуске многопоточного процесса в системе с одним процессором потоки работают поочередно. Пример работы процессов в многозадачном режиме мы уже видели на рис. 2.1. Иллюзия параллельной работы нескольких различных последовательных процессов создается путем постоянного переключения системы между процессами. Многопоточность реализуется примерно так же. Процессор быстро переключается между потоками, создавая впечатление параллельной работы потоков, хотя и на не столь быстром процессоре. В случае трех ограниченных производительностью процессора потоков в одном процессе все потоки

будут работать параллельно, и каждому потоку будет соответствовать виртуальный процессор с быстродействием, равным одной трети быстродействия реального процессора.

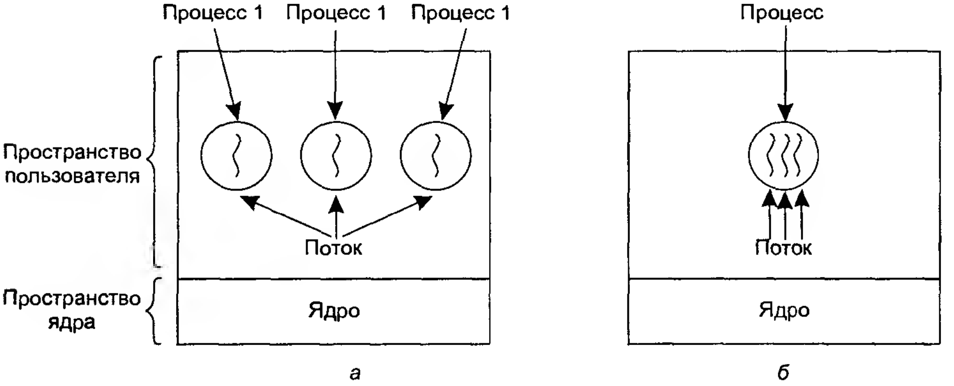


Рис. 2.4. Три процесса с одиночными потоками управления (*а*); один процесс с тремя потоками управления (*б*)

Различные потоки в одном процессе не так независимы, как различные процессы. У всех потоков одно и то же адресное пространство, что означает совместное использование глобальных переменных. Поскольку любой поток имеет доступ к любому адресу ячейки памяти в адресном пространстве процесса, один поток может считывать, записывать или даже стирать информацию из стека другого потока. Защиты не существует, поскольку (1) это невозможно и (2) это ненужно. В отличие от различных процессов, которые могут быть инициированы различными пользователями и преследовать несовместимые цели, один процесс всегда запущен одним пользователем, и потоки созданы таким образом, чтобы работать совместно, не мешая друг другу. Как показано в табл. 2.3, потоки разделяют не только адресное пространство, но и открытые файлы, дочерние процессы, сигналы и т. п. Таким образом, ситуацию на рис. 2.4, *а* следует использовать в случае абсолютно несвязанных процессов, тогда как схема на рис. 2.4, *б* будет уместна, когда потоки выполняют совместно одну работу.

Таблица 2.3. В первой колонке перечислены элементы, совместно используемые всеми потоками процесса, а во второй — элементы, индивидуальные для каждого потока

Элементы процесса	Элементы потока
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и их обработчики	
Информация об использовании ресурсов	

Первая колонка содержит элементы, являющиеся свойствами процесса, а не потока. Например, если один поток открывает файл, этот файл тут же становится видимым для остальных потоков, и они могут считывать информацию и записывать ее в файл. Это логично, поскольку процесс, а не поток является единицей управления ресурсами. Если бы у каждого потока было собственное адресное пространство, открытые файлы, аварийные сигналы, требующие обработки и т. д., это были бы отдельные процессы. Концепция потоков состоит в возможности совместного использования набора ресурсов несколькими потоками для выполнения некой задачи в тесном взаимодействии.

Как и любой обычный процесс (то есть процесс с одним потоком), поток может находиться в одном из нескольких состояний: рабочем, заблокированном, готовности или завершенном. Действующий поток взаимодействует с процессором. Блокированный поток ожидает некоторого события, которое его разблокирует. Например, при выполнении системного запроса чтения с клавиатуры поток блокируется, пока не поступит сигнал с клавиатуры. Поток может быть разблокирован каким-либо внешним событием или другим потоком. Поток в состоянии готовности будет запущен, как только до него дойдет очередь. Переходы между состояниями потоков такие же, как на рис. 2.2.

Важно понимать, что у каждого потока свой собственный стек, как показано на рис. 2.5. Стек каждого потока содержит по одному фрейму для каждой процедуры, вызванной, но еще не вернувшей управления. Во фрейме находятся локальные переменные процедуры и адрес возврата. Например, если процедура *X* вызывает процедуру *Y* и она, в свою очередь, вызывает процедуру *Z*, то во время работы процедуры *Z* в стеке будут находиться фреймы для всех трех процедур. Каждый поток может вызывать различные процедуры и, соответственно, иметь различный протокол выполнения процесса — именно поэтому каждому потоку необходим собственный стек.

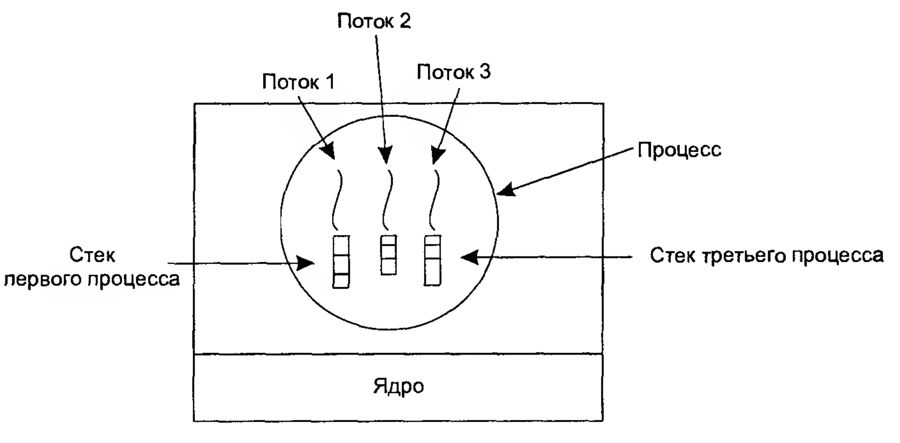


Рис. 2.5. У каждого потока свой собственный стек

В многопоточном режиме процессы, как правило, запускаются с одним потоком. Этот поток может создавать новые потоки, вызывая библиотечную процедуру, например *thread_create*. Параметром обычно является имя процедуры, кото-

рую необходимо запустить для создания нового потока. Указание какой-либо информации, касающейся адресного пространства нового потока, не является необходимым (или даже возможным), поскольку новый поток создается в адресном пространстве существующего потока. Иногда возникает иерархия потоков с отношениями типа «родительский—дочерний поток», но чаще всего иерархия отсутствует и все потоки считаются равнозначными. Независимо от иерархических отношений, создающему потоку чаще всего возвращается идентификатор потока, который дает имя новому потоку.

Выполнив задачу, поток может прекратить работу, вызвав библиотечную процедуру, скажем, *thread_exit*. После этого поток исчезает и уже не рассматривается планировщиком. В некоторых потоковых системах один поток может ждать прекращения работы другого (определенного) потока. Для этого вызывается процедура, например *thread_wait*. Процедура блокирует вызывающий процедуру поток, пока другой поток (определенный) не прекратит работу. В этом отношении создание и завершение потоков очень похожи на создание и завершение процессов с практически такими же параметрами.

Еще одно распространенное обращение потока — *thread_yield* — позволяет потоку добровольно «уступать свою очередь» другому потоку. Это важный момент, поскольку в случае потоков не существует прерывания по таймеру, позволяющего установить режим разделения времени, как это было в случае процессов. Потокам необходимо быть вежливыми и время от времени самим уступать процессор другим потокам. Существуют и процедуры, позволяющие одному потоку подождать, пока другой завершит какое-либо действие, оповестить о том, что он закончил какое-либо действие и т. п.

Несмотря на то что потоки часто бывают полезными, они существенно усложняют программную модель. Представьте себе системный вызов fork в UNIX. Если у родительского процесса было много потоков, должно ли это свойство распространяться на дочерний процесс? Если нет, то процесс может неправильно функционировать, поскольку все потоки могут оказаться необходимы.

Но что произойдет, если поток родительского процесса будет блокирован вызовом read с клавиатуры, а у дочернего процесса столько же потоков, сколько у родительского? Будут ли теперь блокированы два потока — один из родительского процесса, другой из дочернего? И если с клавиатуры поступит строка, получат ли оба потока ее копию? Или только один — тогда какой? Эта же проблема возникает при работе с открытыми сетевыми соединениями.

Другой класс проблем связан с тем, что потоки совместно используют большое количество структур данных. Что произойдет, если один поток закроет файл в то время, когда другой считывает из него данные? Представьте себе, что одному потоку стало недостаточно памяти и он просит выделить дополнительную память. На полпути происходит переключение потоков, и теперь новый поток также замечает, что ему не хватает памяти, и просит выделить дополнительную память. В этой ситуации память может быть выделена дважды. Все эти проблемы можно решить, но для создания корректно работающих многопоточных программ необходима тщательная и всесторонне обдуманная разработка.

Использование потоков

Настало время объяснить, почему же, собственно, потоки так необходимы. Основной причиной является выполнение большинством приложений существенного числа действий, некоторые из них могут время от времени блокироваться. Схему программы можно существенно упростить, если разбить приложение на несколько последовательных потоков, запущенных в квазипараллельном режиме.

С этим рассуждением мы уже сталкивались — оно являлось аргументом в пользу существования процессов, не так ли? Мы можем рассуждать на языке параллельных процессов вместо прерываний, таймеров и переключателей контекста. В случае потоков придется добавить еще один элемент: возможность совместного использования параллельными объектами адресного пространства и всех содержащихся в нем данных. Для определенных приложений эта возможность является существенной, и в таких случаях схема параллельных процессов (с разными адресными пространствами) не подходит.

Еще одним аргументом в пользу потоков является легкость их создания и уничтожения (поскольку с потоком не связаны никакие ресурсы). В большинстве систем на создание потока уходит примерно в 100 раз меньше времени, чем на создание процесса. Это свойство особенно полезно, если необходимо динамическое и быстрое изменение числа потоков.

Третьим аргументом является производительность. Концепция потоков не дает увеличения производительности, если все они ограничены возможностями процессора. Но когда имеется одновременная потребность в выполнении большого объема вычислений и операций ввода-вывода, наличие потоков позволяет совмещать эти виды деятельности во времени, тем самым увеличивая общую скорость работы приложения.

И наконец, концепция потоков полезна в системах с несколькими процессорами, где возможен настоящий параллелизм. Мы еще вернемся к этой теме в главе 8.

Необходимость потоков проще продемонстрировать на конкретных примерах. Возьмем в качестве первого примера текстовый редактор. Большинство текстовых редакторов выводят текст на экран монитора в том виде, в котором он будет напечатан. В частности, разрывы строк и страниц находятся на своих местах, и пользователь может при необходимости их откорректировать (например, удалить неполные строки вверху и внизу страницы, неприемлемые с эстетической точки зрения).

Представьте себе, что пользователь пишет книгу. С точки зрения автора проще всего хранить книгу в одном файле, чтобы легче было искать отдельные разделы, выполнять глобальную замену и т. п. С другой стороны, можно хранить каждую главу в отдельном файле. Но было бы крайне неудобно хранить каждый раздел и параграф в своем файле — в случае глобальных изменений пришлось бы редактировать сотни файлов. Например, если предполагаемый стандарт xxx был утвержден только перед отправкой книги в печать, придется заменять «Черновой стандарт xxx» на «Стандарт xxx» в последнюю минуту. Эта операция делается одной командой в случае одного файла и, напротив, займет очень много времени, если придется редактировать каждый из 300 файлов, на которые разбита книга.

Теперь представьте себе, что произойдет, если пользователь удалит одно предложение на первой странице документа, в котором 800 страниц. Пользователь

перечитал эту страницу и решил исправить предложение на 600-й странице. Он дает команду текстовому редактору перейти на страницу с номером 600 (например, задав поиск фразы, встречающейся только на этой странице). Текстовому редактору придется переформатировать весь документ вплоть до 600 страницы, поскольку до форматирования он не будет знать, где начинается эта страница. Это может занять довольно много времени и вряд ли обрадует пользователя.

В этом случае помогут потоки. Пусть текстовый редактор написан в виде двухпоточной программы. Один поток взаимодействует с пользователем, а второй переформатирует документ в фоновом режиме. Как только предложение на первой странице было удалено, интерактивный поток дает команду фоновому потоку переформатировать весь документ. В то время как первый поток продолжает отслеживать и выполнять команды с клавиатуры или мыши — предположим, прокручивает первую страницу, — второй поток быстро переформатирует книгу. Немного везения — и форматирование будет закончено раньше, чем пользователь захочет перейти к 600 странице, и тогда команда будет выполнена мгновенно.

Раз уж мы об этом задумались, почему бы не добавить третий поток? Большинство текстовых редакторов автоматически сохраняет редактируемый текст раз в несколько минут, чтобы пользователь не лишился плодов работы целого дня в случае аварийного завершения программы, отказа системы или перебоев с питанием. Этим может заниматься третий поток, не отвлекая два оставшихся (рис. 2.6).

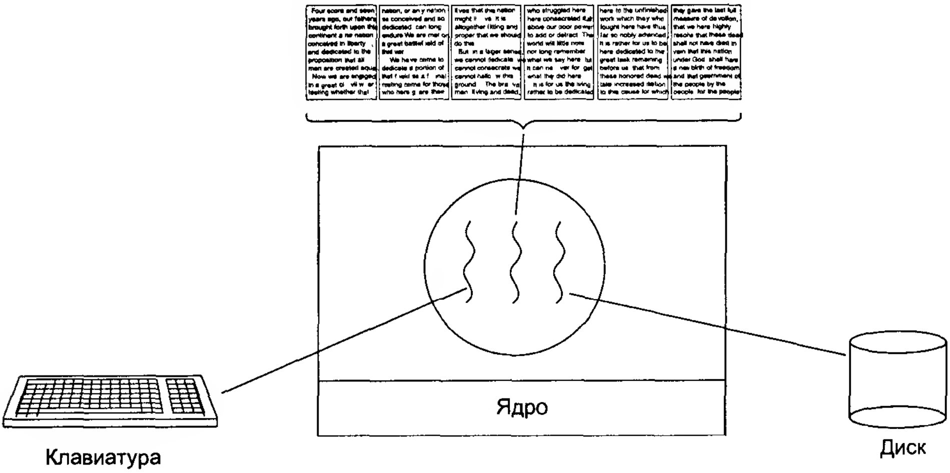


Рис. 2.6. Текстовый редактор с тремя потоками

Если бы программа была однопоточной, тогда при каждой операции сохранения файла все команды с клавиатуры и мыши игнорировались до окончания дисковой операции. У пользователя это создало бы впечатление низкой производительности. В качестве альтернативы команды с клавиатуры и мыши могут прерывать сохранение файла, обеспечивая высокую производительность, но приводя к сложной программной модели, управляемой прерываниями. Программная модель с тремя потоками существенно проще. Первый поток взаимодействует

с пользователем, второй при необходимости переформатирует документ, а третий периодически сохраняет на диске содержимое оперативной памяти.

Очевидно, что в этом случае модель с тремя процессами не подойдет, поскольку всем трем необходимо работать с одним и тем же документом. Три же потока совместно используют общую память, и все три имеют доступ к документу.

Ситуация выглядит точно так же в случае многих других интерактивных программ. Динамическая электронная таблица — программа, позволяющая пользователю работать с матрицей, некоторые элементы которой заданы пользователем. Остальные элементы вычисляются на основе заданных элементов с использованием достаточно сложных формул. Если пользователь изменяет один элемент матрицы, это приводит к пересчету многих других элементов. Пока один поток занят пересчетом элементов в фоновом режиме, другой может позволить пользователю в это время вносить дальнейшие изменения. А третий поток может периодически сохранять резервную копию файла на диске.

Теперь давайте рассмотрим еще одну ситуацию, в которой необходимы потоки: сервер web-сайта. На сервер приходят запросы, и клиенту отсылается содержимое запрашиваемых web-страниц. У большинства web-сайтов некоторые страницы существенно более посещаемы, чем другие. Например, главная страница компании Sony посещается гораздо чаще, чем страница с техническими спецификациями конкретных моделей записывающих видеокамер. Для повышения эффективности работы сервер использует эту особенность, храня содержимое особо популярных страниц в основной памяти (чтобы не надо было каждый раз обращаться за ними на диск). Этот раздел памяти называется **кэш**, и он используется также во многих других ситуациях.

На рис. 2.7 представлен один из способов организации web-сервера. Один поток, называемый **диспетчером**, считывает приходящие по сети запросы. После этого он находит свободный (то есть блокированный) **рабочий поток** и передает ему запрос, скажем, записывая указатель сообщения в специальное слово, связанное с каждым потоком. Затем диспетчер активизирует ждущий поток, переводя его из состояния блокировки в состояние готовности.

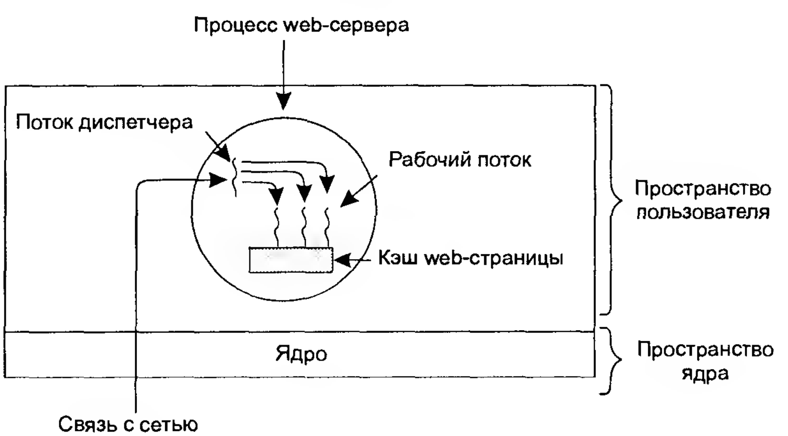


Рис. 2.7. Многопоточный web-сервер

После активации рабочий поток проверяет возможность удовлетворения запроса в кэше web-страниц, к которому имеют доступ все потоки. В случае отрицательного ответа поток начинает операцию чтения read, чтобы считать страницу с диска, и блокируется до завершения этой операции. Когда рабочий поток блокируется, для запуска выбирается следующий поток, им может оказаться диспетчер или другой готовый рабочий поток.

Эта модель позволяет создать сервер в виде набора последовательных потоков. Программа диспетчера состоит из бесконечного цикла, в который входит получение запроса и передача его рабочему потоку. Программа каждого рабочего потока состоит из бесконечного цикла, включающего получение запроса от диспетчера и проверку кэша на наличие запрашиваемой страницы. При наличии страницы в кэше она отсылается клиенту, и рабочий процесс блокируется в ожидании нового запроса. При отсутствии страницы в кэше она считывается с диска, отсылается клиенту, и рабочий процесс блокируется в ожидании нового запроса.

Приблизительный набросок программы представлен на рис. 2.8. Здесь и в дальнейшем *TRUE* предполагается константой, равной 1. Переменные *buf* и *page* являются структурами, подходящими соответственно для хранения запроса и web-страницы.

```
while(TRUE) {
     get_next_request(&buf);
     handoff_work(&buf);
}
```

а

```
while(TRUE) {
       wait_for_work(&buf)
       look_for_page_in_cache(&buf,&page);
       if(page_not_in_cache(&page))
          read_page_from_disk(&buf,&page);
      return_page(&page);
}
```

б

Рис. 2.8. Приблизительный набросок программы для рис. 2.7: поток диспетчера (*а*); рабочий поток (*б*)

Теперь рассмотрим, как можно было написать web-сервер в отсутствие потоков. Одна из возможностей — заставить его работать как один поток. Основной цикл получает запрос, проверяет его и удовлетворяет, затем переходит к следующему. Пока web-страница считывается с диска, сервер простаивает и не обрабатывает другие поступающие запросы. Если сервер расположен на выделенном компьютере — а чаще всего именно так и бывает, — процессор простаивает, пока web-страница считывается с диска. В результате в единицу времени однопоточное приложение может обрабатывать существенно меньшее число запросов. Таким образом, использование нескольких потоков дает заметное увеличение производительности, хотя каждый поток программируется последовательно, обычным способом.

Итак, мы рассмотрели два возможных варианта: web-сервер с одним потоком и несколькими потоками. Представьте себе, что многопоточная система невозможна, но хочется увеличить эффективность системы с одним потоком. Возможен третий вариант web-сервера в случае существования версии системного запроса read без блокировки. На сервер приходит запрос, его считывает и проверяет единственный поток. Если запрашиваемая web-страница есть в кэше — хорошо, если нет — запускается дисковая операция без блокировки.

Сервер записывает в таблицу текущее состояние запроса и переходит к следующему событию. Оно может быть как новым запросом, так и ответом предыдущей операции. В случае нового запроса он начинает обрабатываться, в противном случае соответствующая информация считывается из таблицы и формируется ответ. В случае процедуры ввода-вывода с диска без блокировки ответ может иметь форму сигнала или прерывания.

При такой схеме модель «последовательных процессов», которая была справедлива в первых двух ситуациях, не действует. Состояние программы должно явно сохраняться и восстанавливаться в таблице каждый раз, когда сервер переключается между запросами. Фактически мы имитируем потоки и стеки, причем не самым простым способом. Такая модель, в которой каждому расчету соответствует сохраненное состояние и есть несколько событий, которые могут изменить это состояние, называется машиной с конечным числом состояний или **конечным автоматом**. Эта модель широко используется в программировании.

Теперь должно быть ясно, какие преимущества привносят потоки. Они дают возможность сохранить модель последовательных процессов, выполняющих блокирующие системные запросы (например, для ввода-вывода с диска), и тем не менее добиться параллелизма. Системные запросы с блокировкой упрощают программирование, а параллелизм увеличивает производительность. Однопоточный сервер сохраняет простоту программирования, связанную с наличием блокирующих системных запросов, но уступает в производительности. Модель конечного автомата существенно повышает производительность при помощи параллелизма, но использует системные запросы без блокировки, что усложняет программирование. Эти модели представлены в табл. 2.4.

Таблица 2.4. Три способа конструирования сервера

Модель	Характеристики
Потоки	Параллелизм, системные запросы с блокировкой
Процесс с одним потоком	Нет параллелизма, системные запросы с блокировкой
Конечный автомат	Параллелизм, системные запросы без блокировки, прерывания

Третий пример необходимости потоков — приложения, оперирующие большим количеством данных. Обычно считывается блок данных, обрабатывается и снова записывается. Проблема состоит в том, что при наличии только системных запросов с блокировкой процесс будет блокироваться при чтении и записи данных. Необходимо избегать простоя процессора, особенно при таком большом объеме вычислений.

Решение проблемы — в потоках. Процесс можно разбить на входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные и помещает их во входной буфер. Обрабатывающий поток считывает данные из входного буфера, обрабатывает их и помещает в выходной буфер, а выходной поток считывает их оттуда и записывает обратно на диск. В такой модели считывание данных, обработка и запись происходят одновременно. Разумеется, это возможно лишь в том случае, когда системные вызовы блокируют только вызывающий поток, а не весь процесс.

Реализация потоков в пространстве пользователя

Есть два основных способа реализации пакета потоков: в пространстве пользователя и ядре. Выбор между ними остается спорным вопросом, и возможна смешанная реализация. Мы рассмотрим оба способа, а также их преимущества и недостатки.

Первый метод состоит в размещении пакета потоков целиком в пространстве пользователя. При этом ядро о потоках ничего не знает и управляет обычными, однопоточными процессами. Наиболее очевидное преимущество этой модели состоит в том, что пакет потоков на уровне пользователя можно реализовать даже в операционной системе, не поддерживающей потоки. Все операционные системы когда-то относились к этой категории, а некоторые относятся до сих пор.

Подобные реализации имеют в своей основе одинаковую общую схему, представленную на рис. 2.9, *а*. Потоки работают поверх системы поддержки исполнения программ, которая является набором процедур, управляющих потоками. С четырьмя из них мы уже знакомы: *thread_create, thread_exit, thread_wait* и *thread_yield*, но обычно их больше.

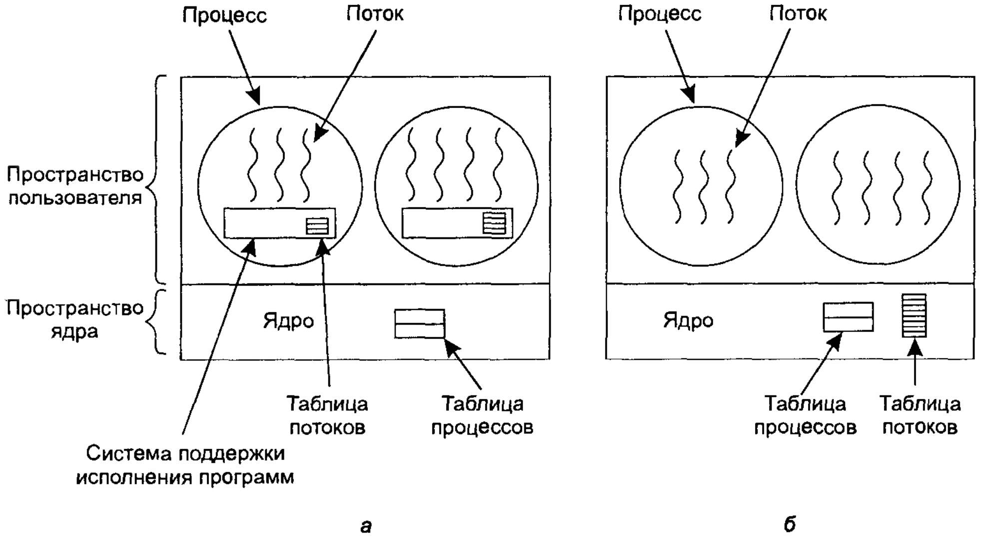


Рис. 2.9. Пакет потоков в пространстве пользователя (*а*); пакет потоков, управляемый ядром (*б*)

Если управление потоками происходит в пространстве пользователя, каждому процессу необходима собственная **таблица потоков** для отслеживания потоков в процессе. Эта таблица аналогична таблице процессов, с той лишь разницей, что она отслеживает лишь характеристики потоков, такие как счетчик команд, указатель вершины стека, регистры, состояние и т. п. Когда поток переходит в состояние готовности или блокировки, вся информация, необходимая для повторного запуска, хранится в таблице потоков подобному тому, как в ядре хранится информация о процессах в таблице процессов.

Когда поток, ожидая окончания действия другого потока в том же процессе, делает нечто, что может привести к локальной блокировке, он вызывает процедуру системы поддержки исполнения программ. Процедура проверяет необходимость блокирования потока. В этом случае процедура сохраняет регистры потока в таблице потоков, ищет в таблице поток, готовый к запуску, и загружает его сохраненные значения в регистры машины. Как только указатель стека и счетчик команд переключены, работа нового потока возобновляется автоматически. Если у процессора есть команда, позволяющая за одну инструкцию сохранить все регистры, и еще одна, чтобы загрузить их все заново, переключение потоков может быть выполнено с помощью очень небольшого количества инструкций. Такое переключение потоков по крайней мере на порядок быстрее, чем переключения в режим ядра, и является серьезным аргументом в пользу управления потоками в пространстве пользователя.

Но существует одно серьезное отличие потоков от процессов. В тот момент, когда поток завершает на время свою работу, например, когда он вызывает процедуру *thread_yield*, программа *thread_yield* может сама сохранить информацию о потоке в таблице потоков. Более того, она может после этого вызвать планировщик потоков для выбора следующего потока. Процедура, сохраняющая информацию о потоке, и планировщик являются локальными процедурами, и их вызов существенно более эффективен, чем вызов ядра. Не требуются прерывание, переключение контекста, сохранение кэша и т. п., что существенно ускоряет переключение потоков.

Потоки, реализованные на уровне пользователя, имеют и другие преимущества. Они позволяют каждому процессу иметь собственный алгоритм планирования. Для некоторых приложений, например приложений с потоком «сборки мусора», оказывается удобным не задумываться о том, что поток может остановиться в неподходящий момент. Эти приложения также лучше масштабируются, поскольку потоки ядра неизменно занимают некоторое пространство в таблице и стековое пространство в ядре, что может стать проблемой в случае большого числа потоков.

Несмотря на более высокую производительность, с реализацией потоков на уровне пользователя связаны и некоторые серьезные проблемы. Первой из них является проблема реализации блокирующих системных запросов. Представьте, что поток начинает считывание с клавиатуры до того, как была нажата хотя бы одна клавиша. Было бы неприемлемо позволить потоку выполнить системный запрос, поскольку это остановило бы все потоки. Одной из основных целей использования потоков было предоставление возможности каждому потоку использовать блокирующие запросы, но так, чтобы один блокированный поток не мешал остальным. Не очень понятно, как достичь этой цели, если использовать блокирующие системные запросы.

Можно сделать все системные запросы не блокирующими (как, например, запрос на чтение с клавиатуры read, который возвращал бы 0 байт в случае отсутствия данных), но это потребует неприемлемых изменений операционной системы. Вспомните, ведь одним из основных аргументов в пользу потоков на уровне пользователя была возможность работы в существующих операционных системах. К тому же изменение семантики запроса read повлекло бы за собой изменения во многих пользовательских программах.

Оказалось, что существует альтернатива, если имеется возможность узнавать заранее, последует ли за запросом блокировка. В некоторых версиях UNIX есть системный запрос select, позволяющий узнать о наличии или отсутствии блокировки у последующего запроса read. При наличии такого системного запроса библиотечную процедуру *read* можно заменить новой, сначала выполняющей запрос select, а потом запрос read, если за ним не последует блокировки. Если блокировка должна произойти, то запрос не выполняется и запускается другой поток. В следующий момент, когда система поддержки исполнения программ получит управление, она может проверить еще раз, последует ли за запросом read блокировка. Такой подход требует перезаписи части библиотеки системных запросов, неэффективен и не отличается изяществом, но выбор не так уж велик. Код, который помещается вокруг системного запроса для проверки на блокировку, называется **чехлом** (jacket) или **упаковкой** (wrapper).

Схожей проблемой является ошибка из-за отсутствия страницы. Мы рассмотрим ее подробнее в главе 4. На данный момент нам важно, что компьютер можно настроить так, чтобы не вся программа находилась в основной памяти. Если программа производит вызов или переход к той инструкции, которой нет в памяти, возникает ошибка из-за отсутствия страницы, и операционная система берет недостающую часть программы с диска. Именно эта ситуация называется ошибкой из-за отсутствия страницы. Процесс блокируется, пока необходимая инструкция не будет найдена и считана. Если поток приводит к ошибке из-за отсутствия страницы, ядро, не знающее о существовании потоков, блокирует весь процесс целиком, до завершения операции чтения с диска, несмотря на наличие остальных нормально функционирующих потоков.

Еще одной проблемой потоков на уровне пользователя является тот факт, что при запуске одного потока ни один другой поток не будет запущен, пока первый поток добровольно не отдаст процессор. Внутри одного процесса нет прерываний по таймеру, в результате чего невозможно создать планировщик для поочередного выполнения потоков. Планировщик ничего не сможет сделать, пока поток не окажется в системе поддержки исполнения программ по собственному желанию.

Одним из решений этой проблемы может стать ежесекундное прерывание, передающее управление системе поддержки исполнения программ, но этот способ достаточно груб и неудобен. Периодические прерывания по таймеру с более высокой частотой не всегда возможны, а если и возможны, то издержки все равно будут существенными. Более того, возможно, что потоку необходимы свои прерывания по таймеру, которые будут конфликтовать с прерываниями системы поддержки исполнения программ.

Еще один, и, возможно, наиболее серьезный аргумент против использования потоков на уровне пользователя состоит в том, что программисты хотят использовать потоки именно в тех приложениях, в которых потоки часто блокируются, например в многопоточном web-сервере. Эти потоки все время посылают системные запросы. И ядру, перехватившему управление, чтобы выполнить системный запрос, не составит труда заодно переключить потоки, если один из них блокирован. При этом исключается необходимость постоянных обращений к системе с запросом select для проверки наличия или отсутствия блокировки у последующего запроса read. А для приложений, которые полностью ограничены возможностями

процессора и редко блокируются, потоки вообще не нужны. Вряд ли кто-либо станет всерьез применять потоки для вычисления первых *n* простых чисел или игры в шахматы.

Реализация потоков в ядре

Теперь рассмотрим ситуацию, в которой ядро знает о существовании потоков и управляет ими. В этом случае система поддержки исполнения программ не нужна, как показано на рис. 2.9, *б*. Нет необходимости и в наличии таблицы потоков в каждом процессе, вместо этого есть единая таблица потоков, отслеживающая все потоки системы. Если потоку необходимо создать новый поток или завершить имеющийся, он выполняет запрос ядра, который создает или завершает поток, внося изменения в таблицу потоков.

Таблица потоков, находящаяся в ядре, содержит регистры, состояние и другую информацию о каждом потоке. Информация та же, что и в случае управления потоками на уровне пользователя, только теперь она располагается не в пространстве пользователя (внутри системы поддержки исполнения программ), а в ядре. Эта информация является подмножеством информации, которую традиционное ядро хранит о каждом из своих однопоточных процессов (то есть подмножеством состояния процесса). Дополнительно ядро содержит обычную таблицу процессов, отслеживающую все процессы системы.

Все запросы, которые могут блокировать поток, реализуются как системные запросы, что требует значительно больших временных затрат, чем вызов процедуры системы поддержки исполнения программ. Когда поток блокируется, ядро по желанию запускает другой поток из этого же процесса (если есть поток в состоянии готовности) либо поток из другого процесса. При управлении потоками на уровне пользователя система поддержки исполнения программ запускает потоки из одного процесса, пока ядро не передает процессор другому процессу (или пока не кончаются потоки, находящиеся в состоянии готовности).

Поскольку создание и завершение потоков в ядре требует относительно больших расходов, некоторые системы используют повторное использование потоков. После завершения поток помечается как нефункционирующий, но в остальном его структура данных, хранящаяся в ядре, не затрагивается. Позже, когда нужно создать новый поток, реактивируется отключенный поток, что позволяет сэкономить на некоторых накладных расходах. При управлении потоками на уровне пользователя повторное использование потоков тоже возможно, но поскольку накладных расходов, связанных с управлением потоками, в этом случае существенно меньше, то и смысла в этом меньше.

Управление потоками в ядре не требует новых не блокирующих системных запросов. Более того, если один поток вызвал ошибку из-за отсутствия страницы, ядро легко может проверить, есть ли в этом процессе потоки в состоянии готовности, и запустить один из них, пока требуемая страница считывается с диска. Основным недостатком управления потоками в ядре является существенная цена системных запросов, поэтому постоянные операции с потоками (создание, завершение и т. п.) приведут к увеличению накладных расходов.

Смешанная реализация

С целью совмещения преимуществ реализации потоков на уровне ядра и на уровне пользователя были опробованы многие способы смешанной реализации. Один из методов заключается в использовании управления ядром и последующем мультиплексировании потоков на уровне пользователя, как показано на рис. 2.10.

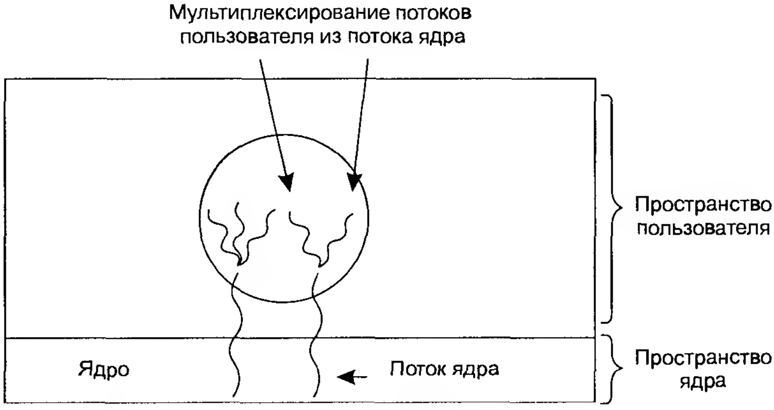


Рис. 2.10. Мультиплексирование потоков пользователя в потоках ядра

В такой модели ядро знает только о потоках своего уровня и управляет ими. Некоторые из этих потоков могут содержать по нескольку потоков пользовательского уровня, мультиплексированных поверх них. Потоки пользовательского уровня создаются, завершаются и управляются так же, как потоки уровня пользователя в процессе, запущенном в не поддерживающей многопоточность системе. Предполагается, что у каждого потока ядра есть набор потоков на уровне пользователя, которые используют его по очереди.

Активация планировщика

Многие исследователи старались совместить преимущества реализации потоков на уровне ядра (простота реализации) и реализации потоков на уровне пользователя (высокая производительность). Ниже мы рассмотрим один из таких подходов, разработанный Андерсоном [12], который называется **активацией планировщика**. Соответствующие разработки описаны в [104, 296].

Целью активации планировщика является имитация функциональности потоков ядра, но с большей производительностью и гибкостью, свойственной потокам уровня пользователя. В частности, пользовательские потоки не должны выполнять специальные системные запросы без блокировки или заранее должны проверять, не вызовет ли запрос блокировку. Тем не менее, когда поток блокируется системным запросом или ошибкой из-за отсутствия страницы, должна оставаться возможность запустить другой поток этого же процесса (если такой есть и находится в состоянии готовности).

Увеличение эффективности достигается за счет уменьшения количества ненужных переходов между пространством пользователя и ядром. Например, если поток блокирован в ожидании действий другого потока, совершенно не обязательно обращаться к ядру, что позволяет избежать накладных расходов по переходу «пользователь—ядро». Система поддержки исполнения программ, работающая в пространстве пользователя, может блокировать синхронизирующий поток и самостоятельно выбрать другой.

При использовании активации планировщика ядро назначает каждому процессу некоторое количество виртуальных процессоров и позволяет системе поддержки исполнения программ (в пространстве пользователя) распределять потоки по процессорам. Этот метод можно использовать и в мультипроцессорной системе, заменяя виртуальные процессоры реальными. Исходное число виртуальных процессоров, соответствующих одному процессу, равно единице, но процесс может запросить больше процессоров и позже вернуть их. Ядро также может забрать виртуальный процессор у одного процесса и отдать другому, более нуждающемуся в нем в данный момент.

В основе механизма работы этой схемы лежит следующее утверждение. Если ядро знает, что поток блокирован (например, если он выполнил блокирующий системный запрос или вызвал ошибку из-за отсутствия страницы), ядро оповещает об этом систему поддержки исполнения программ процесса, пересылая через стек в качестве параметров номер потока в запросе и описание случившегося. Оповещение происходит при помощи активации ядром в определенном начальном адресе системы поддержки исполнения программ, что приблизительно аналогично сигналу в UNIX. Этот метод называется **upcall** («вызов вверх», также иногда именуемый обратным вызовом — callback — в противоположность обычным вызовам, производящимся из верхних уровней в нижние).

Активизированная таким образом система поддержки исполнения программ перепланирует свои потоки, обычно помечая текущий поток как блокированный, выбирая следующий поток из списка, устанавливая значения его регистров и запуская его. Позже, когда ядро получает информацию о том, что поток снова готов к работе (например, канал, из которого он пытался считывать данные, теперь их содержит, или недостающая страница считана с диска), оно выполняет еще один обратный вызов, информируя об этом систему поддержки исполнения программ. Система поддержки исполнения программ по своему усмотрению запускает блокированный поток тут же или помещает его в список готовых процессов, чтобы запустить позже.

При возникновении аппаратного прерывания во время работы потока пользователя процессор переключается в режим ядра. Если прерывание вызвано событием, не имеющим отношения к прерванному процессу, например завершением операции ввода-вывода другого процесса, по завершении работы обработчика прерываний прерванный поток возвращается в состояние, в котором он находился до прерывания. Если же процесс заинтересован в прерывании (например, вызванном поступлением страницы, которую ждал один из потоков процесса), прерванный поток не запускается вновь. Вместо этого прерванный поток приостанавливается, и на этом виртуальном процессоре запускается система поддержки исполнения программ с состоянием прерванного потока на стеке. Дальнейшее зависит от системы

поддержки исполнения программ, решающей, запустить ли на этом процессоре прерванный поток, другой, находящийся в состоянии готовности, или какой-либо третий.

Недостатком метода активации планировщика является существенная зависимость от обратных вызовов, концепция, нарушающая свойственную любой многоуровневой системе структуру. Обычно уровень $n + 1$ может вызывать процедуры уровня n, но не наоборот. Обратные вызовы противоречат этому фундаментальному принципу.

Всплывающие потоки

Потоки часто используются в распределенных системах. Важным примером может служить обработка входящих сообщений, например запросов на обслуживание. Традиционный подход заключается в наличии процесса или потока, который блокируется по системному запросу recieve, ожидая входящего сообщения. Когда сообщение прибывает, оно принимается и обрабатывается.

Возможен и принципиально другой подход, при котором по прибытии сообщения система создает новый поток для его обработки. Такой поток называется **всплывающим**, его схема проиллюстрирована на рис. 2.11. Основным преимуществом всплывающих потоков является их «свежесть» — у такого потока нет истории: регистров, стека и прочей информации, которую нужно восстанавливать. Всплывающие потоки абсолютно «стерильны» и идентичны, что позволяет создавать их быстро. Новый поток обрабатывает входящее сообщение. Использование всплывающих потоков позволяет значительно сократить промежуток времени между прибытием сообщения и началом его обработки.

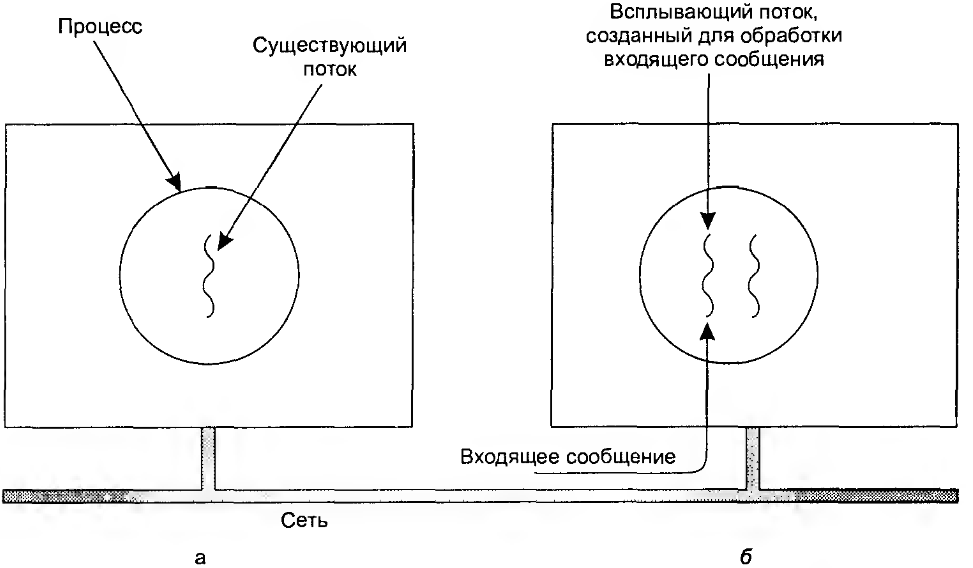


Рис. 2.11. Создание нового потока по прибытии сообщения: до прибытия сообщения (а); после прибытия сообщения (б)

При использовании всплывающих потоков необходимо предварительное планирование. Например, в каком процессе возникнет новый поток? Если система поддерживает потоки, работающие в контексте ядра, новый поток может возникнуть там (именно поэтому мы не показали ядро на рис. 2.11). Создание всплывающих потоков в пространстве ядра всегда быстрее и проще, чем в пространстве пользователя. К тому же всплывающему потоку в пространстве ядра проще получить доступ ко всем таблицам ядра и устройств ввода-вывода, что может оказаться полезным при обработке прерываний. С другой стороны, наличие ошибок в потоке, расположенном в пространстве ядра, может нанести существенно больший ущерб. Например, если поток работает слишком долго и невозможно воспользоваться приоритетным прерыванием, это может привести к потере входных данных.

Как сделать однопоточную программу многопоточной

Многие из существующих программ были написаны для однопоточных процессов. Сделать их многопоточными гораздо сложнее, чем это может показаться на первый взгляд. Ниже мы рассмотрим некоторые из возможных трудностей.

Прежде всего, программа потока обычно состоит из нескольких процедур, так же как и процесс. У этих процедур могут быть локальные переменные, глобальные переменные и параметры. Проблем с локальными переменными и параметрами не будет, зато проблемы будут с переменными, которые являются глобальными для потока, но не глобальными для всей программы. Эти переменные являются глобальными с точки зрения процедур одного потока (которые ими пользуются, как пользовались бы любыми другими глобальными переменными), но не имеют никакого отношения к другим потокам.

В качестве примера рассмотрим переменную *errno* в UNIX. Если процесс (или поток) выполняет неудачный системный запрос, код ошибки записывается в *errno*. На рис. 2.12 поток 1 выполняет системный запрос access, чтобы узнать, имеет ли он разрешение на доступ к конкретному файлу. Операционная система возвращает ответ в глобальной переменной *errno*. После этого управление возвращается к потоку 1. Однако прежде, чем у него появляется возможность считать значение *errno*, планировщик решает, что поток 1 уже достаточно попользовался процессором и пора переключиться на поток 2. Поток 2 выполняет запрос open, завершающийся неудачей, в результате чего значение *errno* изменяется и предыдущее значение теряется. После того как поток 1 вновь получит управление, он прочитает неверное значение *errno*, и дальнейшие его действия будут неправильными.

Существует несколько различных решений проблемы. Одно из решений — запретить глобальные переменные вообще. Какой бы заманчивой ни была эта идея, она вступит в противоречие с большей частью существующего программного обеспечения. Другое решение — предоставить каждому потоку собственные глобальные переменные, как показано на рис. 2.13. В этом случае конфликт исключается, поскольку у каждого потока будет своя копия *errno* и остальных глобальных переменных. Это решение фактически приводит к появлению новых уровней видимости переменных: переменные, доступные всем процедурам потока (в дополнение

к уже имеющимся уровням видимости переменных, доступных только одной процедуре), и переменные, доступные всей программе.

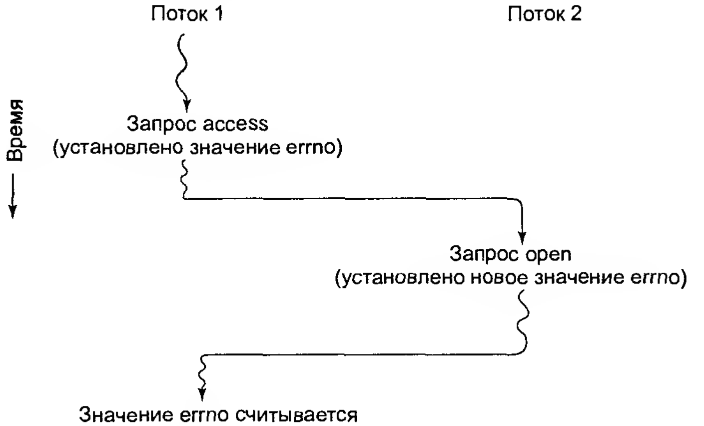


Рис. 2.12. Конфликт между потоками при использовании глобальной переменной

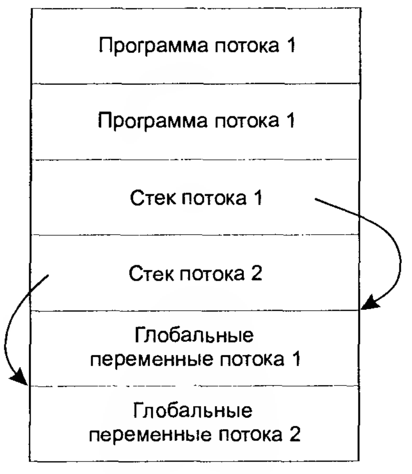


Рис. 2.13. У потоков могут быть собственные глобальные переменные

Обеспечить доступ к собственным глобальным переменным не очень просто, поскольку в большинстве языков программирования есть способы описания локальных и глобальных переменных, но не промежуточных разновидностей. Можно отвести под глобальные переменные отдельный участок памяти и рассматривать их как дополнительные параметры процедур. Несмотря на некоторую неуклюжесть, этот метод работает.

В качестве альтернативы можно написать новые библиотечные процедуры, которые будут создавать, записывать и считывать переменные, глобальные для потока. Первый запрос будет выглядеть примерно так:

```
create_global("bufptr");
```

Этот запрос отводит участок памяти под указатель, называющийся *bufptr*, в динамической памяти или в отдельном участке памяти, зарезервированном для вызывающего потока. Не имеет значения, где именно расположен этот участок памяти, важно, что лишь вызывающий поток имеет к нему доступ. Если другой поток создаст глобальную переменную с таким же именем, она будет размещаться в другом участке памяти и конфликта потоков не будет.

Для доступа к глобальной переменной нужно два запроса: один, чтобы записать ее значение, и другой — чтобы его считать. Для записи будет использоваться что-то вроде

```
set_global("bufptr",&buf);
```

Этот запрос сохраняет значение указателя в участке памяти, созданном запросом *create_global*. Запрос на чтение может выглядеть как

```
bufptr=read_global("bufptr");
```

Запрос возвращает адрес для доступа к данным, хранящийся в глобальной переменной.

Другим препятствием может стать тот факт, что большинство библиотечных процедур не являются реентерабельными. Это означает, что при их написании не предполагалась ситуация, при которой процедуре будет необходимо ответить на второй запрос, не закончив ответа на первый. Например, пересылку сообщения по сети можно организовать следующим образом: сообщение помещается в буфер, затем эмулируется прерывание в ядро для его отсылки. Что произойдет, если один поток поместит сообщение в буфер, а затем прерывание по таймеру приведет к передаче управления второму потоку, который тут же поместит в этот буфер свое сообщение?

Подобная же проблема возникает с процедурами распределения памяти (*malloc* в UNIX), управляющими таблицами использования памяти (в виде связного списка доступных участков памяти). Пока процедура *malloc* занята переписыванием таблиц, таблицы могут временно находиться в несовместимом состоянии, с указателями, никуда не указывающими. Если в этот момент произойдет переключение потоков и от нового потока придет запрос, может быть использован неправильный указатель, что приведет к нарушению работы программы. Решение всех подобных проблем равнозначно полному переписыванию библиотеки.

Другим решением может быть снабжение каждой процедуры чехлом (jacket), устанавливающим бит, означающий, что эта процедура используется. Любая попытка использования процедуры другим потоком до окончания выполнения предыдущего запроса блокируется. Этот метод можно использовать, но он практически исключает параллелизм.

Теперь рассмотрим сигналы. Одни из них связаны с потоками, тогда как другие — нет. Например, если поток выполняет запрос `alarm`, результирующий сигнал по логике должен вернуться к этому потоку. Однако если потоки реализованы в пространстве пользователя, ядро ничего не знает об их существовании и вряд ли направит сигнал к правильному потоку. Ситуация еще больше усложняется, если одновременно у процесса может быть только один необработанный аварийный сигнал, а несколько потоков выполняют запрос `alarm` независимо друг от друга.

Другие сигналы, такие как прерывание с клавиатуры, не связаны с потоками. Кто должен их перехватывать? Один назначенный поток? Все потоки? Специально созданный всплывающий поток? Что случится, если один поток изменит обработчик сигнала, не сообщив об этом остальным потокам? А что если один поток хочет перехватить определенный сигнал (например, CTRL+C с клавиатуры), а другому потоку этот сигнал нужен, чтобы прервать процесс? Подобная ситуация может возникнуть, если один или более потоков пользуются стандартными библиотечными процедурами, а остальные — процедурами, написанными пользователем. Эти потоки абсолютно несовместимы. Вообще говоря, управлять сигналами даже в однопоточной среде достаточно сложно. При переходе к многопоточному окружению обработка сигналов проще не становится.

Последняя проблема, связанная с потоками, — управление стеками. Во многих системах при переполнении стека процесса ядро автоматически увеличивает его. Если у процесса несколько потоков, стеков тоже должно быть несколько. Если ядро не знает о существовании этих стеков, оно не может их автоматически увеличивать при переполнении. Ядро может даже не связать ошибки памяти с переполнением стеков.

Разумеется, эти проблемы не являются непреодолимыми, но на их примере хорошо видно, что введение потоков в существующую систему невозможно без тщательной и продуманной реконструкции всей системы. По крайней мере, придется изменить семантику системных запросов и переписать библиотеки. И результат ваших трудов должен быть совместим с существующими программами для процессов с одним потоком. Дополнительную информацию о потоках можно найти в [149, 225].

Межпроцессное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой. Например, в конвейере ядра выходные данные первого процесса должны передаваться второму и т. д. по цепочке. Поэтому необходимо правильно организованное взаимодействие между процессами, по возможности не использующее прерываний. В этом разделе мы рассмотрим некоторые аспекты **межпроцессного взаимодействия** (IPC, interprocess communication).

Проблема разбивается на три пункта. Первый мы уже упомянули: передача информации от одного процесса другому. Второй связан с контролем над деятельностью процессов: как гарантировать, что два процесса не пересекутся в критических ситуациях (представьте себе два процесса, каждый из которых пытается завладеть последним мегабайтом памяти). Третий касается согласования действий процессов: если процесс *A* должен поставлять данные, а процесс *B* выводить их на печать, то процесс *B* должен подождать и не начинать печатать, пока не поступят данные от процесса *A*. Мы рассмотрим все три случая в следующем подразделе.

Важно понимать, что два из трех описанных пунктов в равной мере относятся и к потокам. Первый — передача информации — в случае потоков проблемой не является, поскольку у потоков общее адресное пространство (передача информации между потоками с разным адресным пространством уже является проблемой

передачи информации между процессами). Остальные два с тем же успехом касаются потоков: те же проблемы, и те же решения. Мы будем рассматривать эти ситуации в контексте процессов, но имейте в виду, что эти же рассуждения применимы и для потоков.

Состояние состязания

В некоторых операционных системах процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа. Местоположение совместно используемой памяти не влияет на суть взаимодействия и возникающие проблемы. Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: спулер печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный **каталог спулера**. Другой процесс, **демон печати**, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

Представьте, что каталог спулера состоит из большого числа сегментов, пронумерованных 0, 1, 2, ..., в каждом их которых может храниться имя файла. Также есть две совместно используемые переменные: *out*, указывающая на следующий файл для печати, и *in*, указывающая на следующий свободный сегмент. Эти две переменные можно хранить в одном файле (состоящем из двух слов), доступном всем процессам. Пусть в данный момент сегменты с 0 по 3 пусты (эти файлы уже напечатаны), а сегменты с 4 по 6 заняты (эти файлы ждут своей очереди на печать). Более или менее одновременно процессы *A* и *B* решают поставить файл в очередь на печать. Описанная ситуация схематически изображена на рис. 2.14.

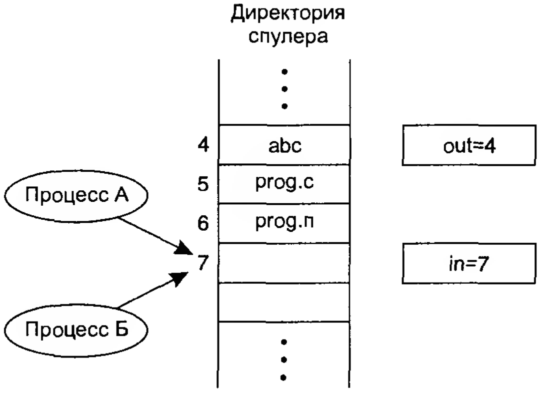


Рис. 2.14. Два процесса хотят одновременно получить доступ к совместно используемой памяти

В соответствии с законом Мерфи (он звучит примерно так: «Если что-то плохое может случиться, оно непременно случится») возможна следующая ситуация.

Процесс *A* считывает значение (7) переменной *in* и сохраняет его в локальной переменной *next_free_slot*. После этого происходит прерывание по таймеру, и процессор переключается на процесс *B*. Процесс *B*, в свою очередь, считывает значение переменной *in* и сохраняет его (опять 7) в своей локальной переменной *next_free_slot*. В данный момент оба процесса считают, что следующий свободный сегмент — седьмой.

Процесс *B* сохраняет в каталоге спулера имя файла и заменяет значение *in* на 8, затем продолжает заниматься своими задачами, не связанными с печатью.

Наконец управление переходит к процессу *A*, и он продолжает с того места, на котором остановился. Он обращается к переменной *next_free_slot*, считывает ее значение и записывает в седьмой сегмент имя файла (разумеется, удаляя при этом имя файла, записанное туда процессом *B*). Затем он заменяет значение *in* на 8 (*next_free_slot* + 1 = 8). Структура каталога спулера не нарушена, так что демон печати не заподозрит ничего плохого, но файл процесса *B* не будет напечатан. Пользователь, связанный с процессом *B*, может в этой ситуации полдня описывать круги вокруг принтера, ожидая требуемой распечатки. Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются **состояниями состязания**. Отладка программы, в которой возможно состояние состязания, вряд ли может доставить удовольствие. Результаты большинства тестовых прогонов будут хорошими, но изредка будет происходить нечто странное и необъяснимое.

Критические области

Как избежать состязания? Основным способом предотвращения проблем в этой и любой другой ситуации, связанной с совместным использованием памяти, файлов и чего-либо еще, является запрет одновременной записи и чтения разделенных данных более чем одним процессом. Говоря иными словами, необходимо **взаимное исключение**. Это означает, что в тот момент, когда один процесс использует разделенные данные, другому процессу это делать будет запрещено. Проблема, описанная в предыдущем параграфе, возникла из-за того, что процесс *B* начал работу с одной из совместно используемых переменных до того, как процесс *A* ее закончил. Выбор подходящей примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы, и мы рассмотрим его подробно в дальнейшем.

Проблему исключения состояний состязания можно сформулировать на абстрактном уровне. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания. В другие моменты времени процесс обращается к совместно используемым данным или выполняет какое-то другое действие, которое может привести к состязанию. Часть программы, в которой есть обращение к совместно используемым данным, называется **критической областью** или **критической секцией**. Если нам удастся избежать одновременного нахождения двух процессов в критических областях, мы сможем избежать состязаний.

Несмотря на то что это требование исключает состязание, его недостаточно для правильной совместной работы параллельных процессов и эффективного использования общих данных. Для этого необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

В абстрактном виде требуемое поведение процессов представлено на рис. 2.15. Процесс *A* попадает в критическую область в момент времени T_1. Чуть позже, в момент времени T_2, процесс *B* пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс *A*, а два процесса не должны одновременно находиться в критических областях. Поэтому процесс *B* временно приостанавливается, до наступления момента времени T_3, когда процесс *A* выходит из критической области. В момент времени T_4 процесс *B* также покидает критическую область, и мы возвращаемся в исходное состояние, когда ни одного процесса в критической области не было.

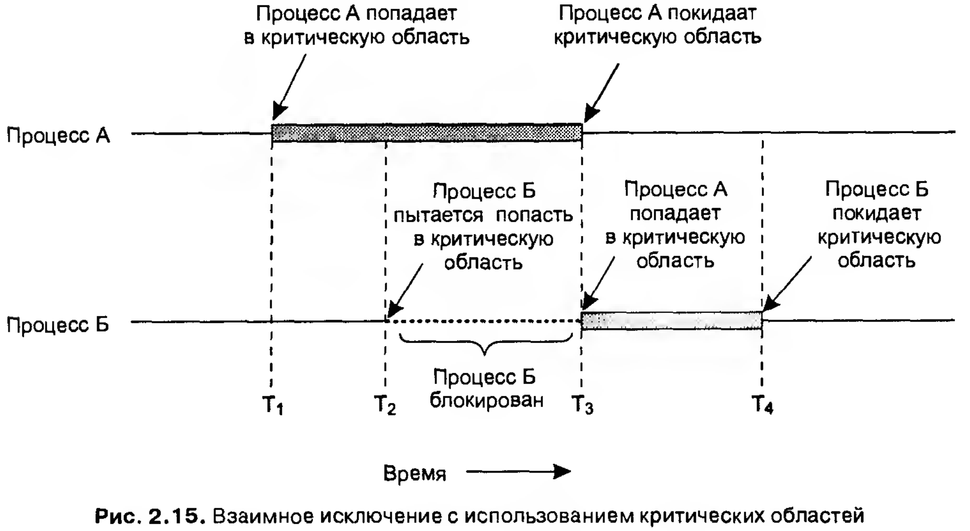


Рис. 2.15. Взаимное исключение с использованием критических областей

Взаимное исключение с активным ожиданием

В этом разделе мы рассмотрим различные способы реализации взаимного исключения с целью избежать вмешательства в критическую область одного процесса при нахождении там другого и связанных с этим проблем.

Запрещение прерываний

Самое простое решение состоит в запрещении всех прерываний при входе процесса в критическую область и разрешение прерываний по выходе из области. Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку процессор переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу процессора другому процессу. Таким образом, запретив прерывания, процесс может спокойно считывать и сохранять совместно используемые данные, не опасаясь вмешательства другого процесса.

И все же было бы неразумно давать пользовательскому процессу возможность запрета прерываний. Представьте себе, что процесс отключил все прерывания и в результате какого-либо сбоя не включил их обратно. Операционная система на этом может закончить свое существование. К тому же в многопроцессорной системе запрещение прерываний повлияет только на тот процессор, который выполнит инструкцию `disable`. Остальные процессоры продолжат работу и сохранят доступ к разделенным данным.

С другой стороны, для ядра характерно запрещение прерываний для некоторых команд при работе с переменными или списками. Возникновение прерывания в момент, когда, например, список готовых процессов находится в неопределенном состоянии, могло бы привести к состоянию состязания. Итак, запрет прерываний бывает полезным в самой операционной системе, но это решение неприемлемо в качестве механизма взаимного исключения для пользовательских процессов.

Переменные блокировки

Теперь попробуем найти программное решение. Рассмотрим одну совместно используемую переменную блокировки, изначально равную 0. Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки. Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0. Таким образом, 0 означает, что ни одного процесса в критической области нет, а 1 означает, что какой-либо процесс находится в критической области.

К сожалению, у этого метода те же проблемы, что и в примере с каталогом спулера. Представьте, что один процесс считывает переменную блокировки, обнаруживает, что она равна 0, но прежде, чем он успевает изменить ее на 1, управление получает другой процесс, успешно изменяющий ее на 1. Когда первый процесс снова получит управление, он тоже заменит переменную блокировки на 1 и два процесса одновременно окажутся в критических областях.

Можно подумать, что проблема решается повторной проверкой значения переменной, прежде чем заменить ее, но это не так. Второй процесс может получить управление как раз после того, как первый процесс закончил вторую проверку, но еще не заменил значение переменной блокировки.

Строгое чередование

Третий метод реализации взаимного исключения иллюстрирован на рис. 2.16. Этот фрагмент программного кода, как и многие другие в этой книге, написан на C. Язык C был выбран, поскольку практически все существующие операционные системы написаны на C (или C++), а не на Java, Modula 3, Pascal и т. п. Язык C

обладает всеми необходимыми свойствами для написания операционных систем: это мощный, эффективный и предсказуемый язык программирования. Язык Java, например, не является предсказуемым, поскольку у программы, написанной на нем, может в критический момент закончиться свободная память и она вызовет «сборщика мусора» в исключительно неподходящее время. В случае С это невозможно, поскольку в С процедура «сбора мусора» в принципе отсутствует. Сравнительный анализ С, С++, Java и еще четырех языков представлен в [268].

```
while(TRUE) {
      while(turn!=0)       /*loop*/;
      critical_region();
      turn=1;
      noncritical_region();
}
```

а

```
while(TRUE) {
      while(turn!=0)       /*loop*/;
      critical_region();
      turn=0;
      noncritical_region();
}
```

б

Рис. 2.16. Предлагаемое решение проблемы критической области: процесс 0 (*а*); процесс 1 (*б*). В обоих случаях необходимо удостовериться в наличии точки с запятой, ограничивающей цикл `while`

На рис. 2.16 целая переменная *turn*, изначально равная 0, отслеживает, чья очередь входить в критическую область. Вначале процесс 0 проверяет значение *turn*, считывает 0 и входит в критическую область. Процесс 1 также проверяет значение *turn*, считывает 0 и после этого входит в цикл, непрерывно проверяя, когда же значение *turn* будет равно 1. Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. Подобного способа следует избегать, поскольку он является бесцельной тратой времени процессора. Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания. Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Когда процесс 0 покидает критическую область, он изменяет значение *turn* на 1, позволяя процессу 1 попасть в критическую область. Предположим, что процесс 1 быстро покидает свою критическую область, так что оба процесса теперь находятся вне критической области, и значение *turn* равно 0. Теперь процесс 0 выполняет весь цикл быстро, выходит из критической области и устанавливает значение *turn* равным 1. В этот момент значение *turn* равно 1, и оба процесса находятся вне критической области.

Неожиданно процесс 0 завершает работу вне критической области и возвращается к началу цикла. Но войти в критическую область он не может, поскольку значение *turn* равно 1 и процесс 1 находится вне критической области. Процесс 0 зависнет в своем цикле `while`, ожидая, пока процесс 1 изменит значение turn на 0. Получается, что метод поочередного доступа к критической области не слишком удачен, если один процесс существенно медленнее другого.

Эта ситуация нарушает третье из сформулированных нами условий: один процесс блокирован другим, не находящимся в критической области. Возвратимся к примеру с каталогом спулера: если заменить критическую область процедурой считывания и записи в каталог спулера, процесс 0 не сможет послать файл на печать, поскольку процесс 1 занят чем-то другим.

Фактически этот метод требует, чтобы два процесса попадали в критические области строго по очереди. Ни один из них не сможет попасть в критическую область (например, послать файл на печать) два раза подряд. Хотя этот алгоритм и исключает состояния состязания, его нельзя рассматривать всерьез, поскольку он нарушает третье условие успешной работы двух параллельных процессов с совместно используемыми данными.

Алгоритм Петерсона

Датский математик Деккер (T. Dekker) был первым, кто разработал программное решение проблемы взаимного исключения, не требующее строгого чередования. Подробное изложение алгоритма можно найти в [46].

В 1981 году Петерсон (G. L. Peterson) разработал существенно более простой алгоритм взаимного исключения. С этого момента алгоритм Деккера стал считаться устаревшим. Алгоритм Петерсона, представленный в листинге 2.1, состоит из двух процедур, написанных на ANSI C, что предполагает необходимость прототипов для всех определяемых и используемых функций. В целях экономии места мы не будем приводить прототипы для этого и последующих примеров.

Листинг 2.1. Решение Петерсона для взаимного исключения

```
#define FALSE 0
#define TRUE  1
#define N     2                      /* Количество процессов */
int turn;                            /* Чья сейчас очередь? */
int interested[N];                   /* Все переменные изначально равны 0 (FALSE) */
void enter_region(int process);      /* Процесс 0 или 1 */
{
      int other;                     /* Номер второго процесса */
      other = 1 - process;           /* Противоположный процесс */
      interested[process] = TRUE;    /* Индикатор интереса*/
      turn = process;                /* Установка флага*/
      while (turn == process && interested[other] == TRUE) /* Пустой оператор */;
}
void leave_region(int process)       /* process: процесс, покидающий критическую область */
{
      interested[process] = FALSE; /* Индикатор выхода из критической области */
}
```

Прежде чем обратиться к совместно используемым переменным (то есть перед тем, как войти в критическую область), процесс вызывает процедуру *enter_region* со своим номером (0 или 1) в качестве параметра. Поэтому процессу при необходимости придется подождать, прежде чем входить в критическую область. После выхода из критической области процесс вызывает процедуру *leave_region*, чтобы обозначить свой выход и тем самым разрешить другому процессу вход в критическую область.

Рассмотрим работу алгоритма более подробно. Исходно оба процесса находятся вне критических областей. Процесс 0 вызывает *enter_region*, задает элементы массива и устанавливает переменную *turn* равной 0. Поскольку процесс 1 не заинтересован в попадании в критическую область, процедура возвращается. Теперь, если процесс 1 вызовет *enter_region*, ему придется подождать, пока *interested* [0] примет значение *FALSE*, а это произойдет только в тот момент, когда процесс 0 вызовет процедуру *leave_region*, чтобы покинуть критическую область.

Представьте, что оба процесса вызвали *enter_region* практически одновременно. Оба сохранят свои номера в *turn*. Сохранится номер того процесса, который был вторым, а предыдущий номер будет утерян. Предположим, что вторым был процесс 1, так что значение *turn* равно 1. Когда оба процесса дойдут до оператора while, процесс 0 войдет в критическую область, а процесс 1 останется в цикле и будет ждать, пока процесс 0 выйдет из критической области.

Команда TSL

Рассмотрим решение, требующее участия аппаратного обеспечения. Многие компьютеры, особенно разработанные с расчетом на несколько процессоров, имеют команду

```
TSL RX,LOCK
```

(Test and Set Lock — проверить и заблокировать), которая действует следующим образом. В регистр RX считывается содержимое слова памяти *lock*, а в ячейке памяти *lock* сохраняется некоторое ненулевое значение. Гарантируется, что операция считывания слова и сохранения неделима — другой процесс не может обратиться к слову в памяти, пока команда не выполнена. Процессор, выполняющий команду TSL, блокирует шину памяти, чтобы остальные процессоры не могли обратиться к памяти.

Воспользуемся командой TSL. Пусть совместно используемая переменная *lock* управляет доступом к разделенной памяти. Если значение переменной *lock* равно 0, любой процесс может изменить его на 1 и обратиться к разделенной памяти, и затем изменить его обратно на 0, пользуясь обычной командой move.

Как использовать эту команду для взаимного исключения? Решение приведено в листинге 2.2. Здесь представлена подпрограмма из четырех команд, написанная на фиктивном (но типичном) ассемблере. Первая команда копирует старое значение *lock* в регистр и затем устанавливает значение переменной равное 1. Потом старое значение сравнивается с нулем. Если оно ненулевое, значит, блокировка уже была установлена и проверка начинается сначала. Рано или поздно значение окажется нулевым (это означает, что процесс, находившийся в критической области, вышел из нее), и подпрограмма возвращается, установив блокировку. Программа просто помещает 0 в переменную *lock*. Специальной команды процессора не требуется.

Листинг 2.2. Вход и выход из критической области с помощью команды TSL

```
enter_region:
        TSL REGISTER,LOCK     | значение lock копируется в регистр, значение переменной
                                устанавливается равным 1
        GMP REGISTER,#0       | Старое значение lock сравнивается с нулем
        JNE enter_region      | Если оно ненулевое, значит, блокировка уже была установлена,
                                поэтому цикл завершается
        RET                   | Возврат к вызывающей программе, процесс вошел в критическую
                                область

leave_region:
        MOVE LOCK,#0          | Сохранение 0 в переменной lock
        RET
```

Одно решение проблемы критических областей теперь очевидно. Прежде чем попасть в критическую область, процесс вызывает процедуру *enter_region*, которая выполняет активное ожидание вплоть до снятия блокировки, затем она устанавливает блокировку и возвращается. По выходе из критической области процесс вызывает процедуру *leave_region*, помещающую 0 в переменную *lock*. Как и во всех остальных решениях проблемы критической области, для корректной работы процесс должен вызывать эти процедуры своевременно, в противном случае взаимное исключение не удастся.

Примитивы межпроцессного взаимодействия

Оба решения — Петерсона и с использованием команды TSL — корректны, но они обладают одним и тем же недостатком: использованием активного ожидания. В сущности, оба они реализуют следующий алгоритм: перед входом в критическую область процесс проверяет, можно ли это сделать. Если нельзя, процесс входит в тугой цикл, ожидая возможности войти в критическую область.

Этот алгоритм не только бесцельно расходует время процессора, но, кроме этого, он может иметь некоторые неожиданные последствия. Рассмотрим два процесса: *H*, с высоким приоритетом, и *L*, с низким приоритетом. Правила планирования в этом случае таковы, что процесс *H* запускается немедленно, как только он оказывается в состоянии ожидания. В какой-то момент, когда процесс *L* находится в критической области, процесс *H* оказывается в состоянии ожидания (например, он закончил операцию ввода-вывода). Процесс *H* попадает в состояние активного ожидания, но поскольку процессу *L* во время работающего процесса *H* никогда не будет предоставлено процессорное время, у процесса *L* не будет возможности выйти иэ критической области, и процесс *H* навсегда останется в цикле. Эту ситуацию иногда называют **проблемой инверсии приоритета**.

Теперь рассмотрим некоторые примитивы межпроцессного взаимодействия, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется процессорное время. Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов sleep и wakeup. Примитив sleep — системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. У запроса wakeup есть один параметр — процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов — адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

Проблема производителя и потребителя

В качестве примера использования этих примитивов рассмотрим проблему **производителя и потребителя**, также известную как проблема **ограниченного буфера**. Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда. (Можно обобщить задачу на случай *m* производителей и *n* потребителей, но мы рассмотрим случай с одним производителем и одним потребителем, поскольку это существенно упрощает решение.)

Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Это решение кажется достаточно простым, но оно приводит к состояниям состязания, как и пример с каталогом спулера. Нам нужна переменная *count* для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно *N*, программа производителя должна проверить, не равно ли *N* значение *count* прежде, чем поместить в буфер следующую порцию данных. Если значение *count* равно *N*, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение *count*.

Код программы потребителя прост: сначала проверить, не равно ли значение *count* нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение *count*. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в листинге 2.3.

Листинг 2.3. Проблема производителя и потребителя с неустранимым состоянием соревнования

```
#define N 100                           /* Максимальное количество элементов в буфере */
int count = 0;                          /* Текущее количество элементов в буфере */

void producer(void)
{
        int item;

        while (TRUE) {                  /* Повторять вечно */
            item = produce_item();      /* Сформировать следующий элемент */
            if (count == N) sleep();    /* Если буфер полон, уйти в состояние ожидания */
            insert_item(item);          /* Поместить элемент в буфер */
            count = count + 1;          /* Увеличить количество элементов в буфере */
            if (count == 1) wakeup(consumer);       /* Был ли буфер пуст? */
        }
}
void consumer(void)
{
        int item;
        while (TRUE) {                  /* Повторять вечно */
            if (count == 0) sleep();    /* Если буфер пуст, уйти в состояние ожидания */
            item = remove_item( );      /* Забрать элемент из буфера */
            count = count - 1;          /* Уменьшить счетчик элементов в буфере */
            if (count == N - 1) wakeup(producer); /* Был ли буфер полон? */
            consume_item(item);         /* Отправить элемент на печать */
        }
}
```

Для описания на языке С системных вызовов `sleep` и `wakeup` мы представили их в виде вызовов библиотечных процедур. В стандартной библиотеке С их нет, но они будут доступны в любой системе, в которой присутствуют такие системные

вызовы. Процедуры *insert_item* и *remove_item* помещают элементы в буфер и извлекают их оттуда.

Теперь давайте вернемся к состоянию состязания. Его возникновение возможно, поскольку доступ к переменной *count* не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной *count*, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение *count*, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова *wakeup*.

Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению *count*, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

Суть проблемы в данном случае состоит в том, что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает. Если бы не это, проблемы бы не было. Быстрым решением может быть добавление **бита ожидания активизации**. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации.

Несмотря на то что введение бита ожидания запуска спасло положение в этом примере, легко сконструировать ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Мы можем добавить еще один бит, или 8, или 32, но это не решит проблему.

Семафоры

В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее [96]. Им был предложен новый тип переменных, так называемые **семафоры**, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.

Дейкстра предложил две операции, down и up (обобщения sleep и wakeup). Операция down сравнивает значение семафора с нулем. Если значение семафора больше нуля, операция down уменьшает его (то есть расходует один из сохраненных сигналов активации) и просто возвращает управление. Если значение семафора равно нулю, процедура down не возвращает управление процессу, а процесс переводится в состояние ожидания. Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое **элементарное действие**. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции. Элементарность операции чрезвычайно важна для разрешения проблемы синхронизации и предотвращения состояния состязания.

Операция up увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию down, один из них выбирается системой (например, случайным образом) и ему разрешается завершить свою операцию down. Таким образом, после операции up, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным 0, но число ожидающих процессов уменьшится на единицу. Операция увеличения значения семафора и активизации процесса тоже неделима. Ни один процесс не может быть блокирован во время выполнения операции up, как ни один процесс не мог быть блокирован во время выполнения операции wakeup в предыдущей модели.

В оригинале Дейкстра использовал вместо down и up обозначения P и V соответственно. Мы не будем в дальнейшем использовать оригинальные обозначения, поскольку тем, кто не знает датского языка, эти обозначения ничего не говорят (да и тем, кто знает язык, говорят немного). Впервые обозначения down и up появились в языке Algol 68.

Решение проблемы производителя и потребителя с помощью семафоров

Как показано в листинге 2.4, проблему потерянных сигналов запуска можно решить с помощью семафоров. Очень важно, чтобы они были реализованы неделимым образом. Стандартным способом является реализация операций down и up в виде системных запросов, с запретом операционной системой всех прерываний на период проверки семафора, изменения его значения и возможного перевода процесса в состояние ожидания. Поскольку для выполнения всех этих действий требуется всего лишь несколько команд процессора, запрет прерываний не приносит никакого вреда. Если используются несколько процессоров, каждый семафор необходимо защитить переменной блокировки с использованием команды TSL, чтобы гарантировать одновременное обращение к семафору только одного процессора. Необходимо понимать, что использование команды TSL принципиально отличается от активного ожидания, при котором производитель или потребитель ждут наполнения или опустошения буфера. Операция с семафором займет несколько микросекунд, тогда как активное ожидание может затянуться на существенно больший промежуток времени.

Листинг 2.4. Проблема производителя и потребителя с семафорами

```
#define N 100                     /* количество сегментов в буфере */
typedef int semaphore;            /* семафоры - особый вид целочисленных переменных */
semaphore mutex = 1;              /* контроль доступа в критическую область */
semaphore empty = N;              /* число пустых сегментов буфера */
semaphore full = 0;               /* число полных сегментов буфера */

void producer(void)
{
	int item;

	while (TRUE) {            /* TRUE - константа, равная 1*/
	    item = produce_item(); /* создать данные, помещаемые в буфер */
	    down(&empty);          /* уменьшить счетчик пустых сегментов буфера */
```

продолжение

Листинг 2.4 (*продолжение*)

```
            down(&mutex):                   /* вход в критическую область */
            insert_item(item);              /* поместить в буфер новый элемент */
            up(&mutex):                     /* выход из критической области */
            up(&full);                      /* увеличить счетчик полных сегментов буфера */
        }
}

void consumer(void)
{
        int item:
        while (TRUE) {                      /* бесконечный цикл */
            down(&full);                    /* уменьшить числа полных сегментов буфера */
            down(&mutex);                   /* вход в критическую область */
            item = remove_item():           /* удалить элемент из буфера */
            up(&mutex);                     /* выход из критической области */
            up(&empty);                     /* увеличить счетчик пустых сегментов буфера */
            consume_item(item);             /* обработка элемента */
        }
}
```

В представленном решении используются три семафора: один для подсчета заполненных сегментов буфера (*full*), другой для подсчета пустых сегментов (*empty*), а третий предназначен для исключения одновременного доступа к буферу производителя и потребителя (*mutex*). Значение счетчика *full* исходно равно нулю, счетчик *empty* равен числу сегментов в буфере, а *mutex* равен 1. Семафоры, исходное значение которых равно 1, используемые для исключения одновременного нахождения в критической области двух процессов, называются **двоичными семафорами**. Взаимное исключение обеспечивается, если каждый процесс выполняет операцию down перед входом в критическую область и up после выхода из нее.

Теперь, когда у нас есть примитивы межпроцессного взаимодействия, вернемся к последовательности прерываний, показанной в табл. 2.2. В системах, использующих семафоры, естественным способом скрыть прерывание будет связать с каждым устройством ввода-вывода семафор, исходно равный нулю. Сразу после запуска устройства ввода-вывода управляющий процесс выполняет операцию down на соответствующем семафоре, тем самым входя в состояние блокировки. В случае прерывания обработчик прерывания выполняет up на соответствующем семафоре, переводя процесс в состояние готовности. В такой модели пятый шаг в табл. 2.2 заключается в выполнении up на семафоре устройства, чтобы следующим шагом планировщик смог запустить программу, управляющую устройством. Разумеется, если в этот момент несколько процессов находятся в состоянии готовности, планировщик может выбрать другой, более значимый процесс. Мы рассмотрим некоторые алгоритмы планирования позже в этой главе.

В примере, представленном в листинге 2.4, семафоры использовались двумя различными способами. Это различие достаточно значимо, чтобы сказать о нем особо. Семафор *mutex* используется для реализации взаимного исключения, то есть для исключения одновременного обращения к буферу и связанным переменным двух процессов. Мы рассмотрим взаимное исключения и методы его реализации в следующем разделе.

Остальные семафоры использовались для **синхронизации**. Семафоры *full* и *empty* необходимы, чтобы гарантировать, что определенные последовательности событий происходят или не происходят. В нашем случае они гарантируют, что производитель прекращает работу, когда буфер полон, а потребитель прекращает работу, когда буфер пуст.

Мьютексы

Иногда используется упрощенная версия семафора, называемая мьютексом (mutex, сокращение от mutual exclusion — взаимное исключение). Мьютекс не способен считать, он может лишь управлять взаимным исключением доступа к совместно используемым ресурсам или кодам. Реализация мьютекса проста и эффективна, что делает использование мьютексов особенно полезным в случае потоков, действующих только в пространстве пользователя.

Мьютекс — переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном. Поэтому для описания мьютекса требуется всего один бит, хотя чаще используется целая переменная, у которой 0 означает неблокированное состояние, а все остальные значения соответствуют блокированному состоянию. Значение мьютекса устанавливается двумя процедурами. Если поток (или процесс) собирается войти в критическую область, он вызывает процедуру *mutex_lock*. Если мьютекс не заблокирован (то есть вход в критическую область разрешен), запрос выполняется и вызывающий поток может попасть в критическую область.

Напротив, если мьютекс заблокирован, вызывающий поток блокируется до тех пор, пока другой поток, находящийся к критической области, не выйдет из нее, вызвав процедуру *mutex_unlock*. Если мьютекс блокирует несколько потоков, то из них случайным образом выбирается один.

Мьютексы легко реализовать в пользовательском пространстве, если доступна команда TSL. Код программы для процедур *mutex_lock* и *mutex_unlock* в случае потоков на уровне пользователя представлен в листинге 2.5.

Листинг 2.5. Реализация mutex_lock и mutex_unlock

```
mutex_lock:
        TSL REGISTER,MUTEX       | Старое значение мьютекса копируется в регистр;
                                   устанавливается новое значение 1
        CMP REGISTER,#0          | Сравнение старого значения с нулем
        JZE ok                   | Если старое значение было нулем, мьютекс не был
                                 | блокирован. Возврат
        CALL thread_yield        | Мьютекс занят, управление передается другому потоку
        JMP mutex_lock           | Повторить попытку позже
ok: RET                          | Возврат, вход в критическую область

mutex_unlock:
        MOVE MUTEX,#0            | Устанавливается значение мьютекса 0
        RET                      | Возврат
```

Процедура *mutex_lock* похожа на процедуру *enter_region* в листинге 2.2, но с одним существенным отличием. Если процедуре *enter_region* не удается войти в критическую область, она продолжает в цикле проверять наличие блокировки (активное

ожидание). В конце концов время, отведенное этому процессу, кончается и планировщик передает управление другому процессу. Раньше или позже процесс, заблокировавший вход в критическую область, освобождает его.

В случае потоков ситуация кардинально меняется, поскольку нет прерываний по таймеру, останавливающих слишком долго работающие потоки. Поток, пытающийся получить доступ к семафору и находящийся в состоянии активного ожидания, зациклится навсегда, поскольку он не позволит предоставить процессор другому потоку, желающему снять блокировку.

В этой ситуации *mutex_lock ведет себя по-другому. Если* войти в критическую область невозможно, *mutex_lock* вызовет *thread_yeld*, чтобы предоставить процессор другому потоку. Активного ожидания здесь нет. При следующем запуске поток снова проверит блокировку.

Поскольку вызов *thread_yeld* является всего лишь обращением к планировщику потоков в пространстве пользователя, он выполняется очень быстро. Следовательно, ни *mutex_lock,* ни *mutex_unlock* не требуют обращений к ядру. Синхронизация потоков на уровне пользователя происходит полностью в пространстве пользователя, с применением процедур, состоящих всего из нескольких команд процессора.

Система мьютексов, которую мы только что рассмотрели, является только скелетом набора запросов. Программное обеспечение часто требует реализации разнообразных возможностей, и примитивы синхронизации не являются исключением. Например, в некоторых реализациях пакета потоков поставляется вызов *mutex_trylock*, который либо предоставляет доступ к критической области, либо возвращает код ошибки, но в любом случае мгновенно возвращает управление, то есть не заставляет поток ждать. Этот запрос дает потоку возможность выбора в случае наличия альтернативы простому ожиданию.

Одну тему мы до сих пор обходили стороной, хотя стоило бы по крайней мере прояснить ее. В случае потоков в пользовательском пространстве нет проблемы доступа потоков к мьютексу, поскольку у всех потоков общее адресное пространство. Тем не менее в большинстве предыдущих моделей, в частности в алгоритме Петерсона и семафорах, молчаливо предполагалось, что несколько процессов имеют доступ к совместно используемому участку памяти, пусть содержащему одно слово. Если адресные пространства процессов несовместны, как мы постоянно утверждали, как они могут совместно использовать переменную *turn* в алгоритме Петерсона, или семафоры, или общий буфер?

На этот вопрос существует два ответа. Во-первых, некоторые из совместно используемых структур данных, скажем, семафоры, могут храниться в ядре с доступом только через системные запросы. Этот подход решает проблему. Во-вторых, большинство современных операционных систем (включая UNIX и Windows) предоставляют возможность совместного использования процессами некоторой части адресного пространства. В этом случае возможно разделение буфера и других структур данных. В крайнем случае, можно совместно использовать файл.

Если два или больше процессов разделяют частично или полностью адресные пространства, различие между процессами и потоками частично размывается, но тем не менее все равно остается. Два процесса с общим адресным пространством все равно обладают разными открытыми файлами, аварийными таймерами и прочими характеристиками, присущими процессам, в то время как два потока, разде-

ляющие адресное пространство, разделяют и все остальное. И в любом случае несколько процессов, совместно использующих адресное пространство, никогда не будут столь же эффективны, как потоки на уровне пользователя, поскольку управление потоками всегда происходит через ядро.

Мониторы

Межпроцессное взаимодействие с применением семафоров выглядит довольно просто, не правда ли? Эта простота кажущаяся. Взгляните внимательнее на порядок выполнения процедур down перед помещением или удалением элементов из буфера в листинге 2.4. Представьте себе, что две процедуры down в программе производителя поменялись местами, так что значение *mutex* было уменьшено раньше, чем *empty*. Если буфер был заполнен, производитель блокируется, установив *mutex* на 0. Соответственно, в следующий раз, когда потребитель обратится к буферу, он выполнит down с переменной *mutex*, равной 0, и тоже заблокируется. Оба процесса заблокированы навсегда. Эта неприятная ситуация называется взаимоблокировкой, и мы вернемся к ней в главе 3.

Вышеизложенная ситуация показывает, с какой аккуратностью нужно обращаться с семафорами. Одна маленькая ошибка, и все останавливается. Это напоминает программирование на ассемблере, но на самом деле еще сложнее, поскольку такие ошибки приводят к абсолютно невоспроизводимым и непредсказуемым состояниям состязания, взаимоблокировкам и т. п.

Чтобы упростить написание программ, в 1974 году Хоар (Hoare) [155] и Бринч Хансен (Brinch Hansen) [43] предложили примитив синхронизации более высокого уровня, называемый **монитором**. Их предложения несколько отличались друг от друга, как мы увидим дальше. Монитор — набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора. В листинге 2.6 представлен монитор, написанный на воображаемом языке Pidgin Pascal.

Листинг 2.6. Монитор

```
monitor example
        integer i;
        condition c;

        procedure producer();
        .
        .
        .
        end;

        procedure consumer();
        .       .       .
        end;
end monitor;
```

Реализации взаимных исключений способствует важное свойство монитора: при обращении к монитору в любой момент времени активным может быть только один процесс. Мониторы являются структурным компонентом языка программи-

рования, поэтому компилятор знает, что обрабатывать вызовы процедур монитора следует иначе, чем вызовы остальных процедур. Обычно при вызове процедуры монитора первые несколько команд процедуры проверяют, нет ли в мониторе активного процесса. Если активный процесс есть, вызывающему процессу придется подождать, в противном случае запрос удовлетворяется.

Реализация взаимного исключения зависит от компилятора, но обычно используется мьютекс или бинарный семафор. Поскольку взаимное исключение обеспечивает компилятор, а не программист, вероятность ошибки гораздо меньше. В любом случае программист, пишущий код монитора, не должен задумываться о том, как компилятор организует взаимное исключение. Достаточно знать, что, обеспечив попадание в критические области через процедуры монитора, можно не бояться попадания в критическую область двух процессов одновременно.

Хотя мониторы предоставляют простой способ реализации взаимного исключения, этого недостаточно. Необходим также способ блокировки процессов, которые не могут продолжать свою деятельность. В случае проблемы производителя и потребителя достаточно просто поместить все проверки буфера на заполненность и пустоту в процедуры монитора, но как процесс заблокируется, обнаружив полный буфер?

Решение заключается во введении **переменных состояния** и двух операций, `wait` и `signal`. Когда процедура монитора обнаруживает, что она не в состоянии продолжать работу (например, производитель выясняет, что буфер заполнен), она выполняет операцию `wait` на какой-либо переменной состояния, скажем, *full*. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор.

Другой процесс, в нашем примере потребитель может активизировать ожидающего напарника, например, выполнив операцию `signal` на той переменной состояния, на которой он был заблокирован. Чтобы в мониторе не оказалось двух активных процессов одновременно, нам необходимо правило, определяющее последствия операции `signal`. Хоар предложил запуск «разбуженного» процесса и остановку второго. Бринч Хансен предложил другое решение: процесс, выполнивший `signal`, должен немедленно покинуть монитор. Иными словами, операция `signal` выполняется только в самом конце процедуры монитора. Мы будем использовать это решение, поскольку оно в принципе проще и к тому же легче в реализации. Если операция `signal` выполнена на переменной, с которой связаны несколько заблокированных процессов, планировщик выбирает и «оживляет» только один из них.

Кроме этого, существует третье решение, не основывающееся на предположениях Хоара и Бринча Хансена: позволить процессу, выполнившему `signal`, продолжать работу и запустить ждущий процесс только после того, как первый процесс покинет монитор.

Переменные состояния не являются счетчиками. В отличие от семафоров они не аккумулируют сигналы, чтобы впоследствии воспользоваться ими. Это означает, что в случае выполнения операции `signal` на переменной состояния, с которой не связано ни одного блокированного процесса, сигнал будет утерян. Проще говоря, операция `wait` должна выполняться прежде, чем `signal`. Это правило существенно упрощает реализацию. На практике это правило не создает проблем, поскольку отслеживать состояния процессов при необходимости не очень трудно. Процесс,

который собирается выполнить signal, может оценить необходимость этого действия по значениям переменных.

В листинге 2.7 представлена схема решения проблемы производителя и потребителя с применением мониторов, написанная на воображаемом языке Pidgin Pascal. В данной ситуации этот язык удобен своей простотой, а также тем, что он позволяет в точности следовать модели Хоара и Бринча Хансена. В каждый момент времени активна только одна процедура монитора. Буфер состоит из *N* сегментов.

Листинг 2.7. Схема решения проблемы производителя и потребителя с применением мониторов

```
monitor ProducerConsumer
	condition full, empty;
	integer count;

	procedure insert(item: integer);
	begin
		if count = N then wait(full);
		insert_item(item);
		count := count+1;
		if count = 1 then signal(empty)
	end;
	function remove: integer;
	begin
		if count = 0 then wait(empty);
		remove = remove_item;
		count := count-1;
		if count = N-1 then signal(full)
	end;
	count := 0;
end monitor;

procedure producer;
begin
	while true do
	begin
		item = produce_item;
		ProducerConsumer.insert(item)
	end
end;

procedure consumer;
begin
	while true do
	begin
		item = ProducerConsumer.remove;
		consume_item(item)
	end
end;
```

Можно подумать, что операции wait и signal похожи на sleep и wakeup, которые приводили к неустранимым состояниям соревнования. Они действительно похожи, но с одним существенным отличием: неудачи применения операций sleep и wakeup были связаны с тем, что один процесс пытался уйти в состояние ожидания, в то время как другой процесс пытался активировать его. С мониторами такого произойти не может. Автоматическое взаимное исключение, реализуемое процедурами

монитора, гарантирует: если производитель, находящийся в мониторе, обнаружит полный буфер и решит выполнить операцию wait, можно не опасаться, что планировщик передаст управление потребителю раньше, чем операция wait будет завершена. Потребитель даже не сможет попасть в монитор, пока операция wait не будет выполнена и производитель не прекратит работу.

Несмотря на то что Pidgin Pascal — воображаемый язык, существует несколько языков программирования, поддерживающих мониторы, хотя и не всегда в соответствии с моделью Хоара и Бринча Хансена. Один из таких языков — Java, объектно-ориентированный язык, поддерживающий потоки на уровне пользователя и позволяющий группировать методы (процедуры) в классы. Добавление в описание метода ключевого слова synchronized гарантирует, что если хотя бы один поток начал выполнение этого метода, ни один другой поток не сможет выполнять другой синхронизированный (определенный как synchronized) метод из этого класса.

Решение проблемы производителя и потребителя с использованием мониторов, написанное на Java, представлено в листинге 2.8. Решение состоит из четырех классов. Внешний класс, *ProducerConsumer*, создает и запускает два потока, *p* и *c*. Второй и третий классы, *producer* и *consumer* соответственно, содержат программы производителя и потребителя. Класс *out_monitor* является монитором. Он содержит два синхронизированных потока, используемых для текущего помещения элементов в буфер и извлечения их оттуда. В отличие от предыдущих примеров, здесь приведен полный текст программ *insert* и *remove*.

Листинг 2.8. Решение проблемы производителя и потребителя на Java

```
public class ProducerConsumer {
      static final int N = 100;                    // константа, задающая размер буфера
      static producer p = new producer();          // создать экземпляр потока производителя
      static consumer c = new consumer();          // создать экземпляр потока потребителя
      static our_monitor mon = new our_monitor();  // создать экземпляр монитора

      public static void main(String args[]) {
         p.start();                                // запуск потока производителя
         c.start();                                // запуск потока потребителя
      }

      static class producer extends Thread {
         public void run() {      // метод run содержит программу потока
            int item;
            while (true) {        // цикл производителя
               item = produce_item();
               mon.insert(item);
            }
         }
         private int produce_item() { ... }      // собственно производство
      }

      static class consumer extends Thread {
         public void run() {      // метод run содержит программу потока
            int item;
            while (true) {        // цикл потребителя
               item = mon.remove();
               consume_item (item);
            }
         }
```

```
        private void consume_item(int item) { ... }   // собственно потребление
        }

        static class our_monitor {          // монитор
            private int buffer[] = new int[N];
            private int count = 0, lo = 0, hi = 0;  // счетчики и индексы
            public synchronized void insert(int val) {
                if (count == N) go_to_sleep();  // если буфер полон, уйти в состояние ожидания
                buffer [hi] = val;        // поместить элемент в буфер
                hi = (hi+1)%N;            // следующий сегмент, в который будет помещен элемент
                count = count+1;          // теперь в буфере на один элемент больше
                if (count == 1) notify();   // если потребитель в состоянии ожидания,
                                              активировать его
            }
            public synchronized int remove( ) {
                int val;
                if (count == 0) go_to_sleep();  // если буфер пуст, уйти в состояние ожидания
                val = buffer [lo];    // забрать элемент из буфера
                lo = (lo+1)%N;        // следующий сегмент, из которого заберут элемент
                count = count -1;     // теперь в буфере на один элемент меньше
                if (count == N -1) notify();   // если производитель в состоянии ожидания,
                                                 активировать его
                return val;
            }
            private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
        }
}
```

Потоки производителя и потребителя функционально идентичны соответствующим частям программы предыдущих примеров. В программе производителя есть бесконечный цикл формирования данных и помещения их в общий буфер. В коде потребителя есть бесконечный цикл с изъятием данных из общего буфера и их обработкой.

Интерес для нас представляет класс *our_monitor*, содержащий буфер, переменные администрирования и два метода синхронизации. Когда производитель активен в процедуре *insert*, потребитель не может быть активным в процедуре *remove*, что исключает состояние состязания. Переменная *count* отслеживает количество элементов в буфере, принимая значения от 0 до $N - 1$. Переменная *lo* является индексом следующего сегмента буфера, из которого следует извлечь данные. Переменная *hi* является индексом следующего сегмента буфера, в который следует поместить данные. Разрешена ситуация, в которой $lo = hi$, что означает 0 или N элементов в буфере. Различать эти два случая можно по переменной *count*.

Синхронизированные методы в языке Java отличаются от стандартных мониторов отсутствием переменных состояния. Взамен предлагаются две процедуры, *wait* и *notify*, которые аналогичны `sleep` и `wakeup` с той лишь разницей, что они используются в синхронизированных методах, а это исключает состояния состязания. Теоретически процедура может быть прервана, для чего и служит весь окружающий ее набор программ. Java требует, чтобы исключения обрабатывались явно. В нашем случае просто представьте, что *go_to_sleep* описывает уход в состояние ожидания.

Благодаря автоматизации взаимного исключения применение мониторов сделало параллельное программирование значительно менее подверженным ошибкам, чем применение семафоров. Но и у мониторов тоже есть свои недостатки.

Недаром два примера мониторов, которые мы рассмотрели, были написаны на Pidgin Pascal и Java, а не на C, как все остальные примеры этой книги. Как мы уже говорили, мониторы являются структурным компонентом языка программирования, и компилятор должен их распознавать и организовывать взаимное исключение. В Pascal, C и многих других языках нет мониторов, поэтому странно было бы ожидать от их компиляторов выполнения правил взаимного исключения. И в самом деле, как может отличить компилятор процедуры монитора от остальных?

В этих языках также нет и семафоров, но их легко добавить: нужно всего лишь присоединить к библиотеке две короткие программы, написанные на ассемблере и реализующие системные вызовы up и down. Компиляторы при этом не обязаны знать об их существовании. Разумеется, операционная система должна знать о семафорах, но даже если у вас операционная система с семафорами, вы можете писать программы для нее на C или C++ (или на ассемблере, если вы склонны к мазохизму). Если же у вас операционная система с мониторами, вам необходим язык со встроенными мониторами.

Другая проблема, связанная с мониторами и семафорами, состоит в том, что они были разработаны для решения задачи взаимного исключения в системе с одним или несколькими процессорами, имеющими доступ к общей памяти. Помещение семафоров в разделенную память с защитой в виде команд TSL может исключить состояния состязания. Эти примитивы будут неприменимы в распределенной системе, состоящей из нескольких процессоров с собственной памятью у каждого, связанных локальной сетью. Вывод из всего вышесказанного следующий: семафоры являются примитивами слишком низкого уровня, а мониторы могут использоваться только в некоторых языках программирования. Примитивы не подходят и для реализации обмена информацией между компьютерами — нужно что-то другое.

Передача сообщений

В роли чего-то другого выступает **передача сообщений**. Этот метод межпроцессного взаимодействия использует два примитива: send и receive, которые скорее являются системными вызовами, чем структурными компонентами языка (что отличает их от мониторов и делает похожим на семафоры). Поэтому их легко можно поместить в библиотечные процедуры, например

```
send(destination, &message);
receive(source, &message);
```

Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника (или от любого источника, если это не имеет значения). Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Разработка систем передачи сообщений

С системами передачи сообщений связано большое количество сложных проблем и конструктивных вопросов, которых не возникает в случае семафоров и мониторов. Особенно много сложностей появляется в случае взаимодействия процессов,

происходящих на различных компьютерах, соединенных сетью. Так, сообщение может потеряться в сети. Чтобы избежать потери сообщений, отправитель и получатель договариваются, что при получении сообщения получатель посылает обратно **подтверждение** приема сообщения. Если отправитель не получает подтверждения через некоторое время, он отсылает сообщение еще раз.

Теперь представим, что сообщение получено, но подтверждение до отправителя не дошло. Отправитель пошлет сообщение еще раз, и до получателя оно дойдет дважды. Крайне важно, чтобы получатель мог отличить копию предыдущего сообщения от нового. Обычно проблема решается с помощью помещения порядкового номера сообщения в тело самого сообщения. Если к получателю приходит письмо с номером, совпадающим с номером предыдущего письма, письмо классифицируется как копия и игнорируется. Решение проблемы успешного обмена информации в условиях ненадежной передачи сообщений составляет основу изучения компьютерных сетей. Информацию по этой теме можно найти в [323].

Для систем обмена сообщениями также важен вопрос названий процессов. Необходимо однозначно определять процесс, указанный в запросе send или receive. Кроме того, встает вопрос **аутентификации**: каким образом клиент может определить, что он взаимодействует с настоящим файловым сервером, а не с самозванцем?

Помимо этого, существуют конструктивные проблемы, существенные при расположении отправителя и получателя на одном компьютере. Одной из таких проблем является производительность. Копирование сообщений из одного процесса в другой происходит гораздо медленнее, чем операция на семафоре или вход в монитор. Было проведено множество исследований с целью увеличения эффективности передачи сообщений. В [65], например, предлагалось ограничивать размер сообщения до размеров регистра и передавать сообщения через регистры.

Решение проблемы производителя и потребителя с передачей сообщений

Теперь рассмотрим решение проблемы производителя и потребителя с передачей сообщений и без использования разделенной памяти. Решение представлено в листинге 2.9. Мы предполагаем, что все сообщения имеют одинаковый размер и сообщения, которые посланы, но еще не получены, автоматически помещаются операционной системой в буфер. В этом решении используются *N* сообщений, по аналогии с *N* сегментами в буфере. Потребитель начинает с того, что посылает производителю *N* пустых сообщений. Как только у производителя оказывается элемент данных, который он может предоставить потребителю, он берет пустое сообщение и отсылает назад полное. Таким образом, общее число сообщений в системе постоянно и их можно хранить в заранее заданном участке памяти.

Если производитель работает быстрее, чем потребитель, все сообщения будут ожидать потребителя в заполненном виде. При этом производитель блокируется в ожидании пустого сообщения. Если потребитель работает быстрее, ситуация инвертируется: все сообщения будут пустыми, а потребитель будет блокирован в ожидании полного сообщения.

Листинг 2.9. Решение проблемы производителя и потребителя с использованием N сообщений

```
#define N 100                        /* количество сегментов в буфере */
void producer(void)
{
      int item;
      message m;                     /* буфер для сообщений */
      while (TRUE) {
         item = produce_item();      /* сформировать нечто, чтобы заполнить буфер */
         receive(consumer, &m);      /* ожидание прибытия пустого сообщения */
         build_message(&m, item);    /* сформировать сообщение для отправки */
         send(consumer, &m);         /* отослать элемент потребителю */
      }
}
void consumer(void)
{
      int item, i;
      message m;
      for (i = 0; i < N; i++) send(producer, &m);   /* отослать N пустых сообщений */
      while (TRUE) {
         receive(producer, &m);      /* получить сообщение с элементом */
         item = extract_item(&m);    /* извлечь элемент из сообщения */
         send(producer, &m);         /* отослать пустое сообщение */
         consume_item(item);         /* обработка элемента */
      }
}
```

Передача сообщений может быть реализована по-разному. Рассмотрим способ адресации сообщений. Можно присвоить каждому из процессов уникальный адрес и адресовать сообщение непосредственно процессам. Другой подход состоит в использовании новой структуры данных, называемой **почтовым ящиком**. Почтовый ящик — это буфер для определенного количества сообщений, тип которых задается при создании ящика. При использовании почтовых ящиков в качестве параметров адреса `send` и `receive` задаются почтовые ящики, а не процессы. Если процесс пытается послать сообщение в полный почтовый ящик, ему приходится подождать, пока хотя бы одно сообщение не будет удалено из ящика.

В задаче производителя и потребителя оба они создадут почтовые ящики, достаточно большие, чтобы хранить *N* сообщений. Производитель будет посылать сообщения с данными в почтовый ящик потребителя, а потребитель будет посылать пустые сообщения в почтовый ящик производителя. С использованием почтовых ящиков метод буферизации очевиден: в почтовом ящике получателя хранятся сообщения, которые были посланы процессу-получателю, но еще не получены.

Другим предельным случаем использования почтовых ящиков является принципиальное отсутствие буферизации. При таком подходе, если `send` выполняется раньше, чем `receive`, посылающий процесс блокируется до выполнения `receive`, когда сообщение может быть напрямую скопировано от отправителя к получателю без промежуточной буферизации. Если `receive` выполняется раньше, чем `send`, получающий процесс блокируется до выполнения `send`. Этот метод часто называется **рандеву**, он легче реализуется, чем схема буферизации сообщений, но менее гибок, поскольку отправитель и получатель должны работать в режиме жесткой синхронизации.

Передача сообщений часто используется в системах с параллельным программированием. Характерным примером системы передачи сообщений является **MPI** (Message-Passing Interface — интерфейс передачи сообщений). Более подробную информацию по этому вопросу можно найти в [139, 308].

Барьеры

Последний из рассмотренных нами механизмов синхронизации предназначался скорее для групп процессов, нежели для ситуаций с двумя процессами типа производитель—потребитель. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти в следующую фазу, пока к этому не готовы все остальные процессы. Этого можно добиться, разместив в конце каждой фазы **барьер**. Когда процесс доходит до барьера, он блокируется, пока все процессы не дойдут до барьера. Действие барьера представлено на рис. 2.17.

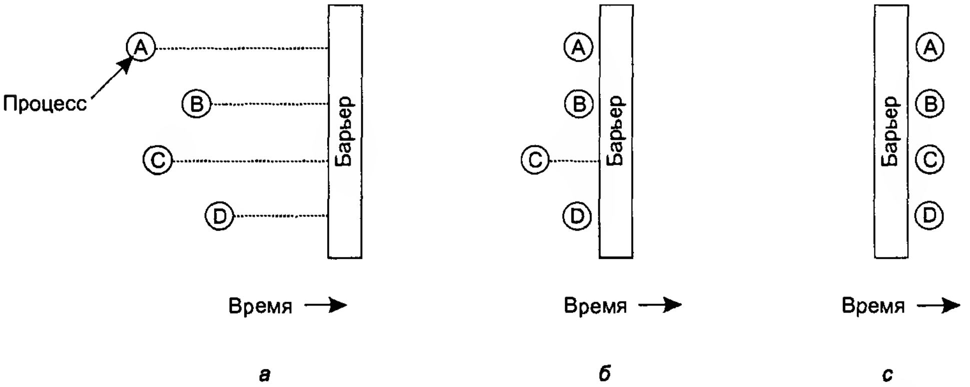


Рис. 2.17. Применение барьеров: процессы, приближающиеся к барьеру (*а*); все процессы, кроме одного, блокированы барьером (*б*); как только последний процесс достигает барьера, все процессы переходят в следующую фазу (*в*)

На рис. 2.17, *а* представлены четыре процесса, приближающиеся к барьеру. Это означает, что они заняты вычислениями и еще не дошли до конца фазы. Через некоторое время первый процесс завершает вычисления, предусмотренные в этой фазе. Он выполняет примитив `barrier`, чаще всего вызывая библиотечную процедуру. Затем процесс приостанавливается. Через некоторое время второй и третий процессы заканчивают первую фазу и выполняют примитив `barrier` (рис. 2.17, *б*). Наконец, когда последний процесс достигает барьера, все процессы переходят в следующую фазу, как показано на рис. 2.17, *в*.

Рассмотрим типичную задачу релаксации в физике или машиностроении в качестве примера ситуации, требующей наличия барьеров. Обычно задача представляется в виде матрицы с некоторыми начальными значениями. Эти значения могут соответствовать температуре в разных точках металлической пластины. Необходимо рассчитать, через какое время установится постоянное распределение температуры в случае нагрева одной стороны пластины.

К матрице применяется некоторое преобразование, например соответствующее законам термодинамики, чтобы получить матрицу значений температуры через

некоторое время. Преобразование применяется снова и снова, чтобы получить зависимость температуры в каждой точке от времени. Результатом будет серия матриц, соответствующих различным моментам времени.

Теперь представим, что матрица очень большая (скажем, миллион на миллион) и для ускорения расчетов необходимы параллельные процессы, возможно, на мультипроцессоре. Различные процессы обрабатывают различные части матрицы, рассчитывая новые элементы на основе старых по законами физики. Очевидно, что ни один процесс не должен начинать итерацию $n + 1$, пока все процессы не закончили свою текущую работу. Этой цели можно достичь, если каждый процесс будет выполнять операцию `barrier`, закончив свою часть итерации. Когда все процессы закончили работу и новая матрица, являющаяся входными данными для следующей итерации, составлена, все процессы одновременно начнут следующую итерацию.

Классические проблемы межпроцессного взаимодействия

Литература по операционным системам содержит множество интересных проблем, которые широко обсуждались и анализировались с применением различных методов синхронизации. В этом разделе мы рассмотрим три наиболее известные проблемы.

Проблема обедающих философов

В 1965 году Дейкстра сформулировал и решил проблему синхронизации, названную им **проблемой обедающих философов**. С тех пор каждый, кто изобретал еще один новый примитив синхронизации, считал своим долгом продемонстрировать достоинства нового примитива на примере проблемы обедающих философов. Проблему можно сформулировать следующим образом: пять философов сидят за круглым столом, и у каждого есть тарелка со спагетти. Спагетти настолько скользкие, что каждому философу нужно две вилки, чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка (рис. 2.18).

Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. (Разумеется, это абстракция, даже применительно к философам, но остальные процессы жизнедеятельности для нашей задачи несущественны.) Когда философ голоден, он пытается получить две вилки, левую и правую, в любом порядке. Если ему удалось получить две вилки, он некоторое время ест, затем кладет вилки обратно и продолжает размышления. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает? (Кое-кто считает, что необходимость двух вилок выглядит несколько искусственно. Возможно, нам следует заменить итальянскую пищу блюдами китайской кухни, спагетти — рисом, а вилки — соответствующими палочками.)

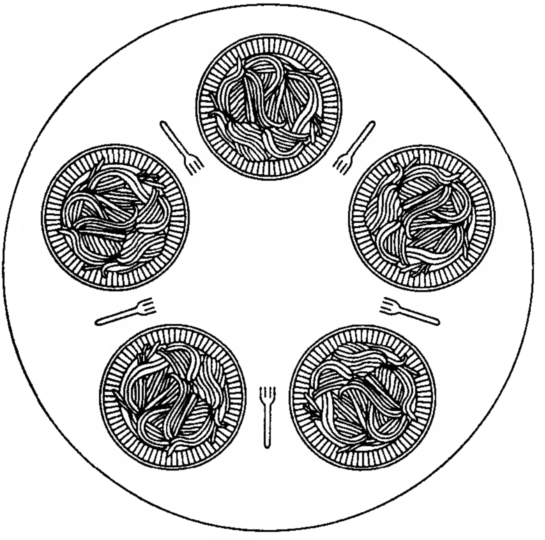


Рис. 2.18. Время обеда на факультете философии

В листинге 2.10 представлено очевидное решение проблемы. Процедура *take_fork* ждет, пока указанная вилка не освободится, и берет ее. К сожалению, это решение неверно — представьте себе, что все пять философов возьмут одновременно свои левые вилки. Каждый останется без правой вилки, и произойдет взаимоблокировка.

Листинг 2.10. Неверное решение проблемы обедающих философов

```
#define N 5                    /* Количество философов */

void philosospher(int i)       /* i — номер философа, от 0 до 4 */
{
      while(TRUE) {
         think();              /* Философ размышляет */
         take_fork(i);         /* Берет левую вилку */
         take_fork((i+1) % N); /* Берет правую вилку */
         eat();                /* Спагетти, ням-ням */
         put_fork(i);          /* Кладет на стол левую вилку */
         put_fork((i+1) % N);  /* Кладет на стол правую вилку */
      }
}
```

Можно изменить программу так, чтобы после получения левой вилки проверялась доступность правой. Если правая вилка недоступна, философ отдает левую обратно, ждет некоторое время и повторяет весь процесс. Этот подход также не будет работать, хотя и по другой причине. Если не повезет, все пять философов могут начать процесс одновременно, взять левую вилку, обнаружить отсутствие правой, положить левую обратно на стол, одновременно взять левую вилку, и так до бесконечности. Ситуация, в которой все программы продолжают работать сколь угодно долго, но не могут добиться хоть какого-то прогресса, называется **зависанием процесса** (по-английски starvation, буквально «умирание от голода». Этот

термин применяется даже в том случае, когда проблема возникает не в итальянском или китайском ресторане, а на компьютерах).

Вы можете подумать: «Если философы будут размышлять в течение некоторого случайно выбранного промежутка времени после неудачной попытки взять правую вилку, вероятность того, что все процессы будут продолжать топтаться на месте хотя бы в течение часа, невелика». Это правильно, и для большинства приложений повторение попытки спустя некоторое время не является проблемой. Например, в локальной сети Ethernet в ситуации, когда два компьютера посылают пакеты одновременно, каждый должен подождать случайно заданное время и повторить попытку — на практике это решение хорошо работает. Тем не менее в некоторых приложениях предпочтительным является другое решение, работающее всегда и не зависящее от случайных чисел (например, в приложении для обеспечения безопасности на атомных электростанциях).

В листинг 2.10 можно внести улучшение, исключающее взаимоблокировку и зависание процесса: защитить пять операторов, следующих за запросом *think*, бинарным семафором. Тогда философ должен будет выполнить операцию down на переменной *mutex* прежде, чем потянуться к вилкам. А после возврата вилок на место ему следует выполнить операцию up на переменной *mutex*. С теоретической точки зрения решение вполне подходит. С точки зрения практики возникают проблемы с эффективностью: в каждый момент времени может есть спагетти только один философ. Но вилок пять, поэтому необходимо разрешить есть в каждый момент времени двум философам.

Решение, представленное в листинге 2.11, исключает взаимоблокировку и позволяет реализовать максимально возможный параллелизм для любого числа философов. Здесь используется массив *state* для отслеживания душевного состояния каждого философа: он либо ест, либо размышляет, либо голодает (пытаясь получить вилки). Философ может начать есть, только если ни один из его соседей не ест. Соседи философа с номером *i* определяются макросами *LEFT* и *RIGHT* (то есть если $i = 2$, то $LEFT = 1$ и $RIGHT = 3$).

Листинг 2.11. Решение задачи обедающих философов

```
#define N         5          /* Количество философов */
#define LEFT      (i+N.1)%N  /* Номер левого соседа философа с номером i */
#define RIGHT     (i+1)%N    /* Номер правого соседа философа с номером i */
#define THINKING  0          /* Философ размышляет */
#define HUNGRY    1          /* Философ пытается получить вилки */
#define EATING    2          /* Философ ест */
typedef int semaphore;       /* Семафоры - особый вид целочисленных переменных */
int state[N];                /* Массив для отслеживания состояния каждого философа */
semaphore mutex = 1;         /* Взаимное исключение для критических областей */
semaphore s[N];              /* Каждому философу по семафору */

void philosopher(int i)      /* i - номер философа. от 0 до N-1 */
{
       while (TRUE) {        /* Повторять до бесконечности */
          think();           /* Философ размышляет */
          take_forks(i);     /* Получает две вилки или блокируется */
          eat();             /* Спагетти. ням-ням */
          put_forks(i);      /* Кладет на стол обе вилки */
       }
}
```

```
void take_forks(int i)          /* i - номер философа, от 0 до N-1*/
{
      down(&mutex);             /* Вход в критическую область */
      state[i] = HUNGRY;        /* Фиксация наличия голодного философа */
      test(i);                  /* Попытка получить две вилки */
      up(&mutex);               /* Выход из критической области */
      down(&s[i]);              /* Блокировка, если вилок не досталось */
}

void put_forks(i)               /* i - номер философа, от 0 до N-1*/
{
      down(&mutex);             /* Вход в критическую область */
      state[i] = THINKING;      /* философ перестал есть */
      test(LEFT);               /* Проверить, может ли есть сосед слева */
      test(RIGHT);              /* Проверить, может ли есть сосед справа */
      up(&mutex);               /* Выход из критической области */
}

void test(i)                    /* i - номер философа, от 0 до N-1*/
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
         state[i] = EATING;
         up(&s[i]);
      }
}
```

В программе используется массив семафоров, по одному на философа, чтобы блокировать голодных философов, если их вилки заняты. Обратите внимание, что каждый процесс запускает процедуру *philosopher* в качестве своей основной программы, но остальные процедуры *take_forks*, *put_forks* и *test* являются обычными процедурами, а не отдельными процессами.

Проблема читателей и писателей

Проблема обедающих философов полезна для моделирования процессов, соревнующихся за монопольный доступ к ограниченному количеству ресурсов, например к устройствам ввода-вывода. Другой известной задачей является проблема читателей и писателей [78], моделирующая доступ к базе данных. Представьте себе базу данных бронирования билетов на самолет, к которой пытается получить доступ множество процессов. Можно разрешить одновременное считывание данных из базы, но если процесс записывает информацию в базу, доступ остальных процессов должен быть прекращен, даже доступ на чтение. Как запрограммировать читателей и писателей? Одно из решений представлено в листинге 2.12.

Листинг 2.12. Решение проблемы читателей и писателей

```
typedef int semaphore;          /* Воспользуйтесь своим воображением */
semaphore mutex = 1;            /* Контроль доступа к rc */
semaphore db = 1;               /* Контроль доступа к базе данных */
int rc = 0;                     /* Количество процессов, читающих или желающих читать */

void reader(void)
{
```

продолжение ⇨

Листинг 2.12 (*продолжение*)

```
        while (TRUE) {                     /* Повторять до бесконечности */
            down(&mutex);                  /* Получение монопольного доступа к rc */
            rc = rc+1;                     /* Одним читающим процессом больше */
            if (rc == 1) down(&db);        /* Если этот читающий процесс - первый… */
            up(&mutex);                    /* Отказ от монопольного доступа к rc */
            read_data_base();              /* Доступ к данным */
            down(&mutex);                  /* Получение монопольного доступа к rc */
            rc = rc-1;                     /* Одним читающим процессом меньше */
            if (rc == 0) up(&db);          /* Если этот читающий процесс - последний... */
            up(&mutex);                    /* Отказ от монопольного доступа к rc */
            use_data_read();               /* Вне критической области */
        }
}

void writer(void)
{
        while (TRUE) {                     /* Повторять до бесконечности */
            think_up_data();               /* Вне критической области */
            down(&db);                     /* Получение монопольного доступа */
            write_data_base();             /* Запись данных */
            up(&db);                       /* Отказ от монопольного доступа */
        }
}
```

Первый читающий процесс выполняет операцию down на семафоре *db*, чтобы получить доступ к базе. Последующие читатели просто увеличивают значение счетчика *rc*. По мере ухода читателей из базы значение счетчика уменьшается, и последний читающий процесс выполняет на семафоре *db* операцию up, позволяя блокированному пишущему процессу получить доступ к базе.

В этом решении один момент требует комментариев. Представьте, что в то время как один читатель уже пользуется базой, другой читатель запрашивает доступ к базе. Доступ разрешается, поскольку читающие процессы друг другу не мешают. Доступ разрешается и третьему, и последующим читателям.

Затем доступ запрашивает пишущий процесс. Запрос отклонен, поскольку пишущим процессам необходим монопольный доступ, и пишущий процесс приостанавливается. Пока в базе есть хотя бы один активный читающий процесс, доступ остальным читателям разрешается, а они все приходят и приходят. Если, предположим, новый читающий процесс запрашивает доступ каждые 2 с, а провести в базе ему надо 5 с, то пишущий процесс никогда в базу не попадет.

Чтобы избежать такой ситуации, нужно немного изменить программу: если пишущий процесс ждет доступа к базе, новый читающий процесс доступа не получает, а становится в очередь за пишущим процессом. Теперь пишущему процессу нужно подождать, пока базу покинут уже находящиеся в ней читающие процессы, но не нужно пропускать вперед читающие процессы, пришедшие к базе после него. Недостаток этого решения заключается в снижении производительности, вызванном уменьшением конкуренции. В [78] представлено решение, в котором пишущим процессам предоставляется более высокий приоритет.

Проблема спящего брадобрея

Действие еще одной классической проблемной ситуации межпроцессного взаимодействия разворачивается в парикмахерской. В парикмахерской есть один брадобрей, его кресло и *n* стульев для посетителей. Если желающих воспользоваться его услугами нет, брадобрей сидит в своем кресле и спит (рис. 2.19). Если в парикмахерскую приходит клиент, он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет). Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания. У этой задачи существует много аналогов в сфере массового обслуживания, например информационная служба, обрабатывающая одновременно ограниченное количество запросов, с компьютеризированной системой ожидания для запросов.

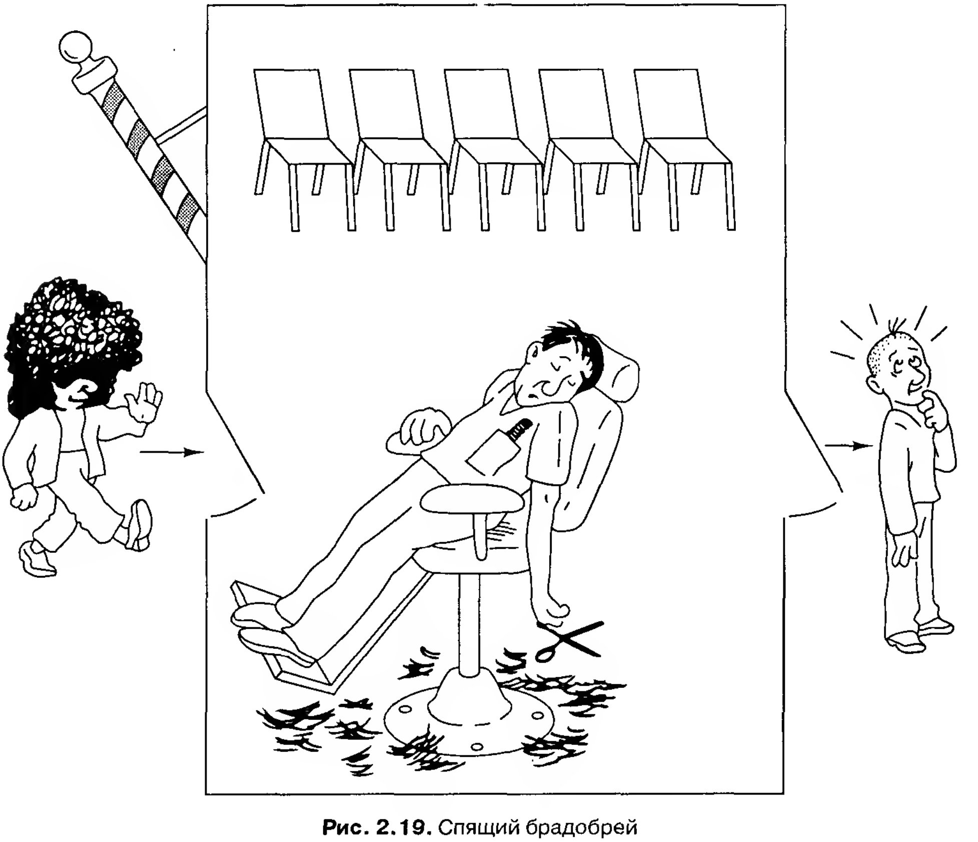


Рис. 2.19. Спящий брадобрей

В предлагаемом решении используются три семафора: *customers*, для подсчета ожидающих посетителей (клиент, сидящий в кресле брадобрея, не учитывается — он уже не ждет); *barbers*, количество брадобреев (0 или 1), простаивающих в ожидании клиента, и *mutex* для реализации взаимного исключения. Также используется переменная *waiting*, предназначенная для подсчета ожидающих посетителей.

Она является копией переменной *customers*. Присутствие в программе этой переменной связано с тем фактом, что прочитать текущее значение семафора невозможно. В этом решении посетитель, заглядывающий в парикмахерскую, должен сосчитать количество ожидающих посетителей. Если посетителей меньше, чем стульев, новый посетитель остается, в противном случае он уходит.

Решение представлено в листинге 2.13. Когда брадобрей приходит утром на работу, он выполняет процедуру *barber*, блокируясь на семафоре *customers*, поскольку значение семафора равно 0. Затем брадобрей засыпает, как показано на рис. 2.19, и спит, пока не придет первый клиент.

Листинг 2.13. Решение проблемы спящего брадобрея

```
#define CHAIRS 5                  /* Количество стульев для посетителей */

typedef int semaphore;            /* Догадайтесь сами */

semaphore customers = 0;          /* Количество ожидающих посетителей */
semaphore barbers = 0;            /* Количество брадобреев, ждущих клиентов */
semaphore mutex = 1;              /* Для взаимного исключения */
int waiting = 0;                  /* Ожидающие (не обслуживаемые) посетители */

void barber(void)
{
      while (TRUE) {
         down(&customers);        /* Если посетителей нет, уйти в состояние ожидания */
         down(&mutex);            /* Запрос доступа к waiting */
         waiting = waiting - 1;   /* Уменьшение числа ожидающих посетителей */
         up(&barbers);            /* Один брадобрей готов к работе */
         up(&mutex);              /* Отказ от доступа к waiting */
         cut_hair();              /* Клиента обслуживают (вне критической области)*/
      }
}

void customer(void)
{
      down(&mutex);               /* Вход в критическую область */
      if (waiting < CHAIRS) {     /* Если свободных стульев нет, придется уйти */
         waiting = waiting + 1;   /* Увеличение числа ожидающих посетителей */
         up(&customers);          /* При необходимости, разбудить брадобрея */
         up(&mutex);              /* Отказ от доступа к waiting */
         down(&barbers);          /* Если брадобрей занят, уйти в состояние ожидания */
         get_haircut();           /* Клиента усаживают и обслуживают */
      } else {
         up(&mutex);              /* Много посетителей, из парикмахерской придется уйти */
      }
}
```

Приходя в парикмахерскую, посетитель выполняет процедуру *customer*, запрашивая доступ к *mutex* для входа в критическую область. Если вслед за ним появится еще один посетитель, ему не удастся что-либо сделать, пока первый посетитель не освободит доступ к *mutex*. Затем посетитель проверяет наличие свободных стульев, в случае неудачи освобождает доступ к *mutex* и уходит.

Если свободный стул есть, посетитель увеличивает значение целочисленной переменной *waiting*. Затем он выполняет процедуру up на семафоре *customers*, тем

самым активизируя поток брадобрея. В этот момент оба — посетитель и брадобрей — активны. Когда посетитель освобождает доступ к *mutex*, брадобрей захватывает его, проделывает некоторые служебные операции и начинает стричь клиента.

По окончании стрижки посетитель выходит из процедуры и покидает парикмахерскую. В отличие от предыдущих программ, цикла посетителя нет, поскольку каждого посетителя стригут только один раз. Цикл брадобрея существует, и брадобрей пытается найти следующего посетителя. Если ему это удается, он стрижет следующего посетителя, в противном случае брадобрей засыпает.

Стоит отметить, что, несмотря на отсутствие передачи данных в проблеме читателей и писателей и в проблеме спящего брадобрея, обе эти проблемы относятся к проблемам межпроцессного взаимодействия, поскольку требуют синхронизации нескольких процессов.

Планирование

Когда компьютер работает в многозадачном режиме, на нем могут быть активными несколько процессов, пытающихся одновременно получить доступ к процессору. Эта ситуация возникает при наличии двух и более процессов в состоянии готовности. Если доступен только один процессор, необходимо выбирать между процессами. Отвечающая за это часть операционной системы называется **планировщиком**, а используемый алгоритм — **алгоритмом планирования**. Рассмотрению задач, связанных с планированием, посвящены следующие разделы.

Многие методы, применимые к планированию процессов, также могут быть применены к планированию потоков, хотя и не все. Для начала мы остановимся на планировании процессов, а позже рассмотрим планирование потоков.

Введение в планирование

Давным-давно, во времена систем пакетной обработки, использовавших отображение содержимого перфокарт на магнитной ленте в качестве устройства ввода, алгоритм планирования был прост: запустить следующую задачу на ленте. С появлением систем с разделением времени алгоритм планирования усложнился, поскольку теперь несколько задач одновременно ожидали обслуживания. На некоторых мэйнфреймах до сих пор совмещаются системы пакетной обработки и службы разделения времени. В результате планировщик должен решать, запустить ли следующим пакетное задание или предоставить процессор интерактивному пользователю за терминалом. (Пакетное задание может предполагать запуск нескольких последовательных программ за один сеанс, но в этом разделе мы считаем, что пакетное задание является запросом на запуск одной программы.) Поскольку время процессора является в такой системе дефицитным ресурсом, хороший планировщик может существенно изменить ощущаемую пользователем производительность работы и, следовательно, степень удовлетворения пользователя. Поэтому много усилий было потрачено на разработку умных и эффективных алгоритмов планирования.

С появлением персональных компьютеров ситуация изменилась. Во-первых, большую часть времени активен только один процесс. Пользователь, работающий с документом в текстовом редакторе, вряд ли будет одновременно считать что-либо в фоновом режиме. Когда пользователь дает команду текстовому процессору, планировщику не приходится долго выбирать, какой процесс запустить, поскольку других кандидатов нет.

Во-вторых, компьютеры стали настолько быстрее, что время процессора практически перестало быть дефицитным ресурсом. Большинство программ для персонального компьютера ограничены скоростью, с которой пользователь вводит входные данные (с клавиатуры или с помощью мыши), а не скоростью процессора. Даже процедуры компиляции, основные потребители процессорного времени прошлого, теперь занимают всего несколько секунд. Если одновременно запущены две программы, например текстовый редактор и электронная таблица, и то вряд ли имеет значение, которая из них была первой, поскольку пользователь, вероятно, ждет окончания работы обеих. Таким образом, на простых персональных компьютерах планирование не играет существенной роли. Разумеется, существуют приложения, занимающие практически весь процессор: визуализация одного часа видеозаписи может потребовать обрабатывающих мощностей промышленного уровня для всех 108 000 кадров формата NTSC (90 000 кадров формата PAL), но подобные приложения являются скорее исключением из правила.

Картина меняется при рассмотрении мощных сетевых рабочих станций и серверов. Здесь планирование снова играет существенную роль, поскольку несколько процессов пытаются получить доступ к процессору. Например, когда процессору нужно выбрать между процессом, перерисовывающим экран после того, как пользователь закрыл окно приложения, и процессом, отсылающим почту, впечатление пользователя о реакции компьютера будет существенно зависеть от этого выбора. Ведь если перерисовка экрана во время отправки почты займет 2 с, пользователь решит, что система очень медленная, тогда как двухсекундная отсылка почты даже не будет замечена. В этом случае планирование процессов очень важно.

Помимо правильного выбора следующего процесса, планировщик также должен заботиться об эффективном использовании процессора, поскольку переключение между процессами требует затрат. Во-первых, необходимо переключиться из режима пользователя в режим ядра. Затем следует сохранить состояние текущего процесса, включая сохранение регистров в таблице процессов, чтобы их можно было загрузить заново позже. В большинстве систем также необходимо сохранить карту памяти (то есть биты-признаки обращения к страницам памяти). Затем нужно выбрать следующий процесс, запустив алгоритм планирования. После этого необходимо перезапустить блок управления памятью с картой памяти нового процесса. И наконец, нужно запустить новый процесс. Помимо этого, переключение между процессами обычно делает бесполезной информацию, содержащуюся в кэше памяти, что приводит к двойной перезагрузке кэша из основной памяти (при входе в ядро и при выходе из ядра). Все это занимает много процессорного времени, особенно при слишком частом переключении между процессами, поэтому в этом рекомендуется соблюдать умеренность.

Поведение процесса

Практически все процессы чередуют периоды вычислений с операциями (дисковыми) ввода-вывода, как показано на рис. 2.20. Обычно процессор некоторое время работает без остановки, затем происходит системный вызов на чтение из файла или запись в файл. После выполнения системного вызова процессор опять считает, пока ему не понадобятся новые данные или не потребуется записать полученные данные и т. д. Заметьте, что некоторые операции ввода-вывода считаются вычислениями. Например, когда процессор копирует биты в видео-ОЗУ, чтобы перерисовать экран, он вычисляет, а не выполняет операцию ввода-вывода, поскольку задействован процессор. С этой точки зрения операция ввода-вывода выполняется в случае, если процесс входит в состояние блокировки, ожидая окончания работы внешнего устройства.

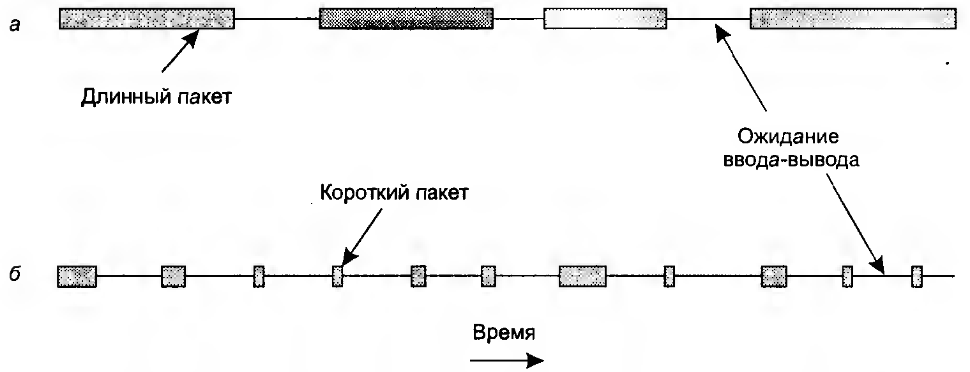


Рис. 2.20. Периоды использования процессора, чередующиеся с ожиданием ввода-вывода: процесс, ограниченный возможностями процессора (*а*); процесс, ограниченный возможностями устройств ввода-вывода (*б*)

Важно отметить, что некоторые процессы, как на рис. 2.20, *а*, большую часть времени заняты вычислениями, а другие, как на рис. 2.20, *б*, большую часть времени ожидают ввода-вывода. Первые называются **ограниченными возможностями процессора**, вторые — **ограниченными возможностями устройств ввода-вывода**. Процессы, ограниченные возможностями процессора, обычно характеризуются длительными периодами использования процессора и нечастыми ожиданиями ввода-вывода, тогда как процессы, ограниченные возможностями устройств ввода-вывода, наоборот: короткими периодами использования процессора и частыми ожиданиями ввода-вывода. Основным параметром здесь является не время ожидания, а время вычислений. Процессы, ограниченные возможностями устройств ввода-вывода, называются так, поскольку загруженность процессора вычислениями между запросами ввода-вывода мала, а не потому, что время выполнения ввода-вывода особо велико. На считывание блока данных с диска уходит одно и то же время, не зависящее от времени, затрачиваемого на обработку этих данных.

Стоит отметить, что с увеличением скорости процессоров процессы становятся все более ограниченными возможностями устройств ввода-вывода. Это связано с тем, что процессоры совершенствуются существенно быстрее, чем диски. Таким образом, планирование процессов, ограниченных возможностями устройств вво-

да-вывода, будет иметь большее значение в будущем. В таких процессах следует избегать простоя относительно медленных дисков.

Когда планировать?

Ключевым вопросом планирования является выбор момента принятия решений. Оказывается, существует множество ситуаций, в которых необходимо планирование. Во-первых, когда создается новый процесс, необходимо решить, какой процесс запустить: родительский или дочерний. Поскольку оба процесса находятся в состоянии готовности, эта ситуация не выходит за рамки обычного и планировщик может запустить любой из двух процессов.

Во-вторых, планирование необходимо, когда процесс завершает работу. Этот процесс уже не существует, следовательно, необходимо из набора готовых процессов выбрать и запустить следующий. Если процессов, находящихся в состоянии готовности, нет, обычно запускается холостой процесс, поставляемый системой.

В-третьих, когда процесс блокируется на операции ввода-вывода, семафоре, или по какой-либо другой причине, необходимо выбрать и запустить другой процесс. Иногда причина блокировки может повлиять на выбор. Например, если *A* — важный процесс и он ожидает выхода процесса *B* из критической области, можно запустить следующим процесс *B*, чтобы он вышел из критической области и позволил процессу *A* продолжать работу. Сложность, однако, в том, что планировщик обычно не обладает информацией, необходимой для принятия правильного решения.

В-четвертых, необходимость планирования может возникнуть при появлении прерывания ввода-вывода. Если прерывание пришло от устройства ввода-вывода, закончившего работу, можно запустить процесс, который был блокирован в ожидании этого события. Планировщик должен выбрать, какой процесс запустить: новый, тот, который был остановлен прерыванием, или какой-то другой.

Если аппаратный таймер выполняет периодические прерывания с частотой 50 Гц, 60 Гц или с любой другой частотой, решения планирования могут приниматься при каждом прерывании по таймеру или при каждом *k*-м прерывании. Алгоритмы планирования можно разделить на две категории согласно их поведению после прерываний. Алгоритмы планирования **без переключений**, иногда называемого также **неприоритетным** планированием, выбирают процесс и позволяют ему работать вплоть до блокировки (в ожидании ввода-вывода или другого процесса), либо вплоть до того момента, когда процесс сам не отдаст процессор. Процесс не будет прерван, даже если он работает часами. Соответственно, решения планирования не принимаются по прерываниям от таймера. После обработки прерывания таймера управление всегда возвращается приостановленному процессу.

Напротив, алгоритмы планирования **с переключениями**, называемого также **приоритетным** планированием, выбирают процесс и позволяют ему работать некоторое максимально возможное фиксированное время. Если к концу заданного интервала времени процесс все еще работает, он приостанавливается и управление переходит к другому процессу (если в очереди есть процесс). Приоритетное планирование требует прерываний по таймеру, происходящих в конце отведенного периода времени, чтобы передать управление планировщику. При отсутствии таймера возможно только планирование без переключений.

Категории алгоритмов планирования

В различных средах требуются различные алгоритмы планирования. Это связано с тем, что различные операционные системы и различные приложения ориентированы на разные задачи. Другими словами, то, для чего следует оптимизировать планировщик, различно в разных системах. Можно выделить три среды:

1. Системы пакетной обработки данных.
2. Интерактивные системы.
3. Системы реального времени.

В системах пакетной обработки нет пользователей, сидящих за терминалами и ожидающих ответа. В таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, отводимым каждому процессу. Такой метод уменьшает количество переключений между процессами и улучшает эффективность.

В интерактивных системах необходимы алгоритмы планирования с переключениями, чтобы предотвратить захват процессора одним процессом. Даже если ни один процесс не захватывает процессор на неопределенно долгий срок намеренно, из-за ошибки в программе один процесс может заблокировать остальные. Для исключения подобных ситуаций используется планирование с переключениями.

В системах с ограничениями реального времени приоритетность, как это ни странно, не всегда обязательна, поскольку процессы знают, что их время ограничено, и быстро выполняют работу, а затем блокируются. Отличие от интерактивных систем в том, что в системах реального времени работают только программы, предназначенные для содействия конкретным приложениям. Интерактивные системы являются универсальными системами. В них могут работать произвольные программы, не сотрудничающие друг с другом и даже враждебные по отношению друг к другу.

Задачи алгоритмов планирования

Чтобы разработать алгоритм планирования, необходимо иметь представление о том, что должен делать хороший алгоритм. Некоторые задачи зависят от среды (системы пакетной обработки, интерактивные или реального времени), но есть задачи, одинаковые во всех системах. Список задач представлен в табл. 2.5. Мы рассмотрим их ниже.

Таблица 2.5. Некоторые задачи алгоритмов планирования

Все системы
Справедливость — предоставление каждому процессу справедливой доли процессорного времени
Принудительное применение политики — контроль за выполнением принятой политики
Баланс — поддержка занятости всех частей системы
Системы пакетной обработки данных
Пропускная способность — максимальное количество задач в час
Оборотное время — минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи
Использование процессора — поддержка постоянной занятости процессора
Интерактивные системы
Время отклика — быстрая реакция на запросы
Соразмерность — выполнение пожеланий пользователя
Системы реального времени
Окончание работы к сроку — предотвращение потери данных
Предсказуемость — предотвращение деградации качества в мультимедийных системах

При всех обстоятельствах необходимо справедливое распределение процессорного времени. Сопоставимые процессы должны получать сопоставимое обслуживание. Выделить одному процессу намного больше времени процессора, чем другому, эквивалентному, несправедливо. Разумеется, с различными категориями процессов можно обращаться весьма по-разному. Возьмите, например, задачи обеспечения безопасности и начисления заработной платы в компьютерном центре атомной электростанции.

С принципом справедливости в какой-то мере связано принудительное применение системной политики. Если локальная политика заключается в предоставлении процессора процессам контроля безопасности по первому требованию, планировщик должен удостовериться в принудительном применении этой политики, даже когда это приводит к начислению заработной платы на 30 с позже.

Еще одной глобальной задачей является контроль занятости всех частей системы. Если устройства ввода-вывода и процессор все время работают, в единицу времени окажется выполнено гораздо больше полезной деятельности, чем если отдельные компоненты будут простаивать. Например, в системах пакетной обработки планировщик выбирает, какие задачи загрузить в память для работы. Гораздо лучше иметь в памяти одновременно несколько процессов, ограниченных возможностями процессора, и несколько процессов, ограниченных возможностями устройств ввода-вывода, чем сначала загрузить и запустить несколько процессов, ограниченных возможностями процессора, и только потом несколько процессов, ограниченных возможностями устройств ввода-вывода. В последнем случае во время работы процессов, ограниченных возможностями процессора, будет простаивать диск, а во время работы процессов, ограниченных возможностями устройств ввода-вывода, будет простаивать процессор. Правильнее будет заставить работать всю систему, аккуратно перемешав процессы.

Руководители крупных компьютерных центров, в которых обрабатываются большие пакетные задания, обычно контролируют три показателя, позволяющие оценить эффективность системы: пропускную способность, оборотное время и использование процессора. **Пропускной способностью** называется число выполненных системой задач в час. В любом случае 50 задач в час лучше, чем 40 задач в час. **Оборотное время** — статистически усредненное время от момента получения задачи до ее выполнения. Оно характеризует время, которое среднестатистический пользователь должен ждать получения выходных данных. Основным правилом является «чем меньше, тем лучше».

Алгоритм планирования, максимизирующий пропускную способность, не обязательно минимизирует оборотное время. При наличии смеси из длинных и коротких задач алгоритм, запускающий только короткие задачи, может добиться высокой пропускной способности (много коротких задач в час), но за счет ужасного оборотного времени для длинных задач. Если короткие задачи поступают с постоянной скоростью, до длинных задач дело может не дойти никогда, в результате чего оборотное время будет бесконечным при высокой пропускной способности.

Учет такой характеристики, как использование процессора, связан с тем, что процессор все еще является самой дорогой частью мэйнфреймов, на которых используются системы пакетной обработки. Руководители таких центров чувствуют себя неловко, если процессор не занят все время. Хотя на самом деле этот показатель не так уж и важен. Гораздо важнее пропускная способность и оборотное время. Рассматривать коэффициент использования процессора в качестве пока-

зателя эффективности примерно так же разумно, как рассматривать рейтинг машин, основанный на количестве оборотов двигателя в минуту.

Для интерактивных систем, особенно систем и серверов, работающих в режиме разделения времени, важны другие задачи. Самой важной является минимизация **времени отклика**, то есть времени между введением команды и получением результата. На персональном компьютере с фоновым процессом (например, отправкой и получением почты) запрос пользователя на запуск программы или открытие файла должен иметь приоритет перед фоновым процессом. Первоочередная обработка всех интерактивных запросов рассматривается как хорошее обслуживание.

Еще одной задачей является достижение **соразмерности**. У пользователя всегда есть собственное (зачастую неправильное) представление о том, сколько времени должна занимать та или иная операция. Если запрос, оцениваемый пользователем как сложный, занимает много времени, пользователь готов с этим согласиться, но если запрос оценивался как простой, пользователь сердится. Так, если после щелчка на значке соединения с Интернет-провайдером, у которого аналоговый модем, до момента выхода в Интернет проходит 45 с, пользователь воспринимает это как должное. Но если 45 с будет занимать отключение от сети, пользователь через 30 с начнет безостановочно ругаться, а к 45-й секунде у него пойдет пена изо рта. Такое поведение пользователя основано на его представлении о том, что звонок по телефону и установление соединения *должно* занимать существенно больше времени, чем разрыв соединения. В некоторых случаях (и в этом тоже) планировщик не может ничего сделать, но может улучшить ситуацию во многих других, особенно если задержка вызвана неправильным порядком процессов.

Системы реального времени обладают другими свойствами, чем интерактивные системы, соответственно и задачи планирования будут разными. Характерной особенностью систем реального времени является наличие жестких сроков выполнения определенных действий. Например, если компьютер управляет устройством, выдающим данные с постоянной скоростью, неспособность своевременно запустить процесс, обрабатывающий данные, приведет к потере данных. Поэтому первоочередной задачей в системах реального времени является строгое соблюдение всех (или большинства) требований по срокам.

В некоторых системах реального времени, особенно связанных с мультимедиа, важна предсказуемость. Небольшая временная задержка не является катастрофичной, но неравномерность аудиопроцесса тут же скажется на ухудшении качества звука. Это касается и изображения, но слух более чувствителен к вибрации, чем зрение. Чтобы исключить эту проблему, планирование процессов необходимо сделать предсказуемым и регулярным. Мы рассмотрим в этой главе алгоритмы планирования для систем пакетной обработки и интерактивных систем, но рассмотрение планирования в системах реального времени отложим до главы 7, в которой изучим мультимедийные операционные системы.

Планирование в системах пакетной обработки данных

Перейдем от общих проблем планирования к конкретным алгоритмам. В этом параграфе мы рассмотрим алгоритмы, используемые в системах пакетной обработки, а в дальнейших параграфах обратимся к алгоритмам интерактивных систем

и систем реального времени. Следует отметить, что некоторые алгоритмы используются как в системах пакетной обработки, так и в интерактивных системах — мы рассмотрим их позже. В этом параграфе мы ограничимся алгоритмами планирования, подходящими только для систем пакетной обработки.

«Первым пришел — первым обслужен»

Алгоритм без переключений «первым пришел — первым обслужен» является, пожалуй, самым простым из алгоритмов планирования. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. Чаще всего формируется единая очередь ждущих процессов. Как только появляется первая задача, она немедленно запускается и работает столько, сколько необходимо. Остальные задачи ставятся в конец очереди. Когда текущий процесс блокируется, запускается следующий в очереди, а когда блокировка снимается, процесс попадает в конец очереди.

Основным преимуществом этого алгоритма является то, что его легко понять и столь же легко программировать. Он справедлив в том же самом смысле, в каком справедливо распределение дефицитных билетов на концерт или соревнования среди всех желающих стоять в очереди с двух часов ночи. В этом алгоритме все процессы в состоянии готовности контролируются одним связным списком. Чтобы выбрать процесс для запуска, нужно всего лишь взять первый элемент списка и удалить его. Появление нового процесса приводит к помещению его в конец списка — что может быть проще?

К сожалению, у этого алгоритма есть существенный недостаток. Представьте себе, что есть один процесс, ограниченный возможностями процессора, который каждый раз работает ровно 1 с, и много процессов, ограниченных возможностями устройств ввода-вывода, каждый из которых очень в небольшой мере использует процессор, но должен выполнить 1000 обращений к диску. Процесс, ограниченный возможностями процессора, запускается, работает 1 с, затем читает блок с диска. Теперь запускаются все процессы ввода-вывода и считывают информацию с диска. Когда процесс, ограниченный возможностями процессора, получает свой блок с диска, он запускается еще на 1 с, а за ним все процессы, ограниченные возможностями устройств ввода-вывода.

Конечный результат будет следующим: каждый из процессов, ограниченных возможностями устройств ввода-вывода, считывает 1 блок данных в секунду, и им потребуется по 1000 с, чтобы закончить работу. Если алгоритм планирования будет прерывать процесс, ограниченный возможностями процессора, раз в 10 мс, процессы, ограниченные возможностями устройств ввода-вывода, закончат за 10 с вместо 1000 с и не очень замедлят работу процесса, ограниченного возможностями процессора.

«Кратчайшая задача — первая»

Рассмотрим еще один алгоритм без переключений для систем пакетной обработки, предполагающий, что временные отрезки работы известны заранее. Например, работники страховой компании могут довольно точно предсказать, сколько времени займет обработка пакета из 1000 исков, поскольку они делают это каждый день. Если в очереди есть несколько одинаково важных задач, планировщик выби-

рает **первой самую короткую задачу**. Посмотрите на рис. 2.21. У нас есть четыре задачи: *A*, *B*, *C* и *D*, со временем выполнения 8, 4, 4 и 4 мин соответственно. Если мы запустим их в данном порядке, оборотное время задачи *A* будет 8 мин, *B* — 12 мин, *C* — 16 мин и *D* — 20 мин, и среднее время будет равно 14 мин.



Рис. 2.21. Пример алгоритма планирования «Кратчайшая задача — первая»: запуск четырех задач в исходном порядке (*а*); запуск в соответствии с алгоритмом (*б*)

Запустим задачи в соответствии с алгоритмом, как показано на рис. 2.21, *б*. Теперь значения оборотного времени составляют 4, 8, 12 и 20 мин соответственно, а среднее время равно 11 мин. Алгоритм оптимизирует задачу. Рассмотрим четыре процесса со временем выполнения *a*, *b*, *c* и *d*. Первая задача выполняется за время *a*, вторая — за время $a + b$ и т. д. Среднее оборотное время будет равно $(4a + 3b + 2c + d)/4$. Очевидно, что вклад времени *a* в среднее больше, чем всех остальных интервалов времени, поэтому первой должна выполняться самая короткая задача, а последней — самая длинная, вносящая вклад, равный собственному оборотному времени. Точно так же рассматривается система из любого количества задач.

Следует отметить, что эта схема работает лишь в случае одновременного наличия задач. В качестве контрпримера можно рассмотреть пять задач, *A*, *B*, *C*, *D* и *E*, причем первые две доступны стразу же, а три оставшиеся — еще через три минуты. Время выполнения этих задач составляет 2, 4, 1, 1 и 1 мин соответственно. Вначале можно выбрать только *A* или *B*, поскольку остальные недоступны. Если руководствоваться алгоритмом «Кратчайшая задача — первая», задачи будут запущены в следующем порядке: *A*, *B*, *C*, *D*, *E*, и среднее оборотное время составит 4,6 мин. Если же запустить их в порядке *B*, *C*, D, E, A, то среднее оборотное время составит 4,4 мин.

Наименьшее оставшееся время выполнения

Версией предыдущего алгоритма с переключениями является алгоритм **наименьшего оставшегося времени выполнения**. В соответствии с этим алгоритмом планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения. В этом случае также необходимо заранее знать время выполнения задач. Когда поступает новая задача, ее полное время выполнения сравнивается с оставшимся временем выполнения текущей задачи. Если время выполнения новой задачи меньше, текущий процесс приостанавливается и управление передается новой задаче. Эта схема позволяет быстро обслуживать короткие запросы.

Трехуровневое планирование

Системы пакетной обработки позволяют реализовать трехуровневое планирование, как показано на рис. 2.22. По мере поступления в систему новые задачи сначала помещаются в очередь, хранящуюся на диске. **Впускной планировщик** выбирает задание и передает его системе. Остальные задания остаются в очереди.

Характерный алгоритм входного контроля может заключаться в выборе смеси из процессов, ограниченных возможностями процессора, и процессов, ограниченных возможностями устройств ввода-вывода. Также возможен алгоритм, в котором устанавливается приоритет коротких задач перед длинными. Впускной планировщик волен придержать некоторые задания во входной очереди, а пропустить задание, поступившее позже остальных.

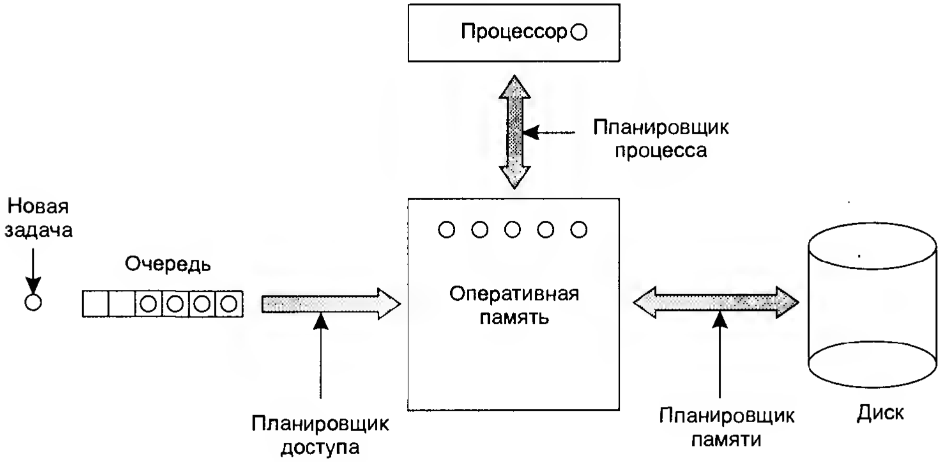


Рис. 2.22. Трехуровневое планирование

Как только задание попало в систему, для него будет создан соответствующий процесс, и он может тут же вступить в борьбу за доступ к процессору. Тем не менее возможна ситуация, когда процессов слишком много и они все в памяти не помещаются, тогда некоторые из них будут выгружены на диск. Второй уровень планирования определяет, какие процессы можно хранить в памяти, а какие — на диске. Этим занимается **планировщик памяти**.

С одной стороны, распределение процессов необходимо часто пересматривать, чтобы у процессов, хранящихся на диске, тоже был шанс получить доступ к процессору. С другой стороны, перемещение процесса с диска в память требует затрат, поэтому к диску следует обращаться не чаще, чем раз в секунду, а может быть и реже. Если содержимое оперативной памяти будет слишком часто меняться, пропускная способность диска будет расходоваться впустую, что замедлит файловый ввод-вывод.

Для оптимизации эффективности системы планировщик памяти должен решить, сколько и каких процессов может одновременно находиться в памяти. Количество процессов, одновременно находящихся в памяти, называется **степенью многозадачности**. Если планировщик памяти обладает информацией о том, какие процессы ограничены возможностями процессора, а какие — возможностями устройств ввода-вывода, он может пытаться поддерживать смесь этих процессов в памяти. Можно грубо оценить, что если некая разновидность процессов будет занимать примерно 20 % времени процессора, то пяти процессов будет достаточно для поддержания постоянной занятости процессора. Более совершенная модель многозадачности будет рассмотрена в главе 4.

Планировщик памяти периодически просматривает процессы, находящиеся на диске, чтобы решить, какой из них переместить в память. Среди критериев, используемых планировщиком, есть следующие:

1. Сколько времени прошло с тех пор, как процесс был выгружен на диск или загружен с диска?
2. Сколько времени процесс уже использовал процессор?
3. Каков размер процесса (маленькие процессы не мешают)?
4. Какова важность процесса?

Третий уровень планирования отвечает за доступ процессов, находящихся в состоянии готовности, к процессору. Когда идет разговор о «планировщике», обычно имеется в виду именно **планировщик процессора**. Этим планировщиком используется любой подходящий к ситуации алгоритм, как с прерыванием, так и без. Некоторые из этих алгоритмов мы уже рассмотрели, а с другими еще ознакомимся.

Планирование в интерактивных системах

Теперь давайте обратимся к алгоритмам планирования, используемым в интерактивных системах. Все они также могут использоваться в качестве планировщика процессора в системах пакетной обработки. В интерактивных системах невозможно трехуровневое планирование, но двухуровневое (планировщик памяти и процессора) возможно и часто используется. Ниже мы рассмотрим алгоритмы для планировщика процессора.

Циклическое планирование

В этом подразделе мы обратимся к некоторым характерным алгоритмам планирования. Одним из наиболее старых, простых, справедливых и часто используемых является алгоритм **циклического планирования**. Каждому процессу предоставляется некоторый интервал времени процессора, так называемый **квант** времени. Если к концу кванта времени процесс все еще работает, он прерывается, а управление передается другому процессу. Разумеется, если процесс блокируется или прекращает работу раньше, переход управления происходит в этот момент. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности согласно рисунку 2.23, *а*. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка (рис. 2.23, *б*).

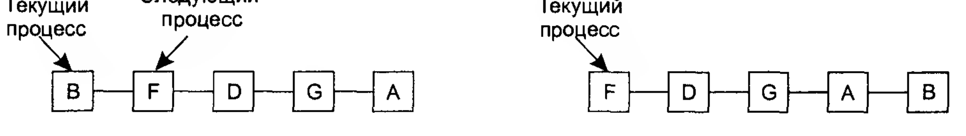


Рис. 2.23. Циклическое планирование: список процессов в состоянии готовности (*а*); список процессов в состоянии готовности после того, как процесс *B* исчерпал свой квант времени (*б*)

Единственным интересным моментом этого алгоритма является длина кванта. Переключение с одного процесса на другой занимает некоторое время — необходимо сохранить и загрузить регистры и карты памяти, обновить таблицы и списки, сохранить и перезагрузить кэш памяти и т. п. Представим, что **переключение процессов** или **переключение контекста**, как его иногда называют, занимает 1 мс, включая переключение карт памяти, перезагрузку кэша и т. п. Пусть размер кванта установлен в 4 мс. В таком случае 20 % процессорного времени уйдет на администрирование — это слишком много.

Для увеличения эффективности назначим размер кванта, скажем, 100 мс. Теперь пропадает только 1 % времени. Но представьте, что будет в системе с разделением времени, если 10 пользователей одновременно нажмут клавишу возврата каретки. В список будет занесено 10 процессов. Если процессор был свободен, первый процесс будет запущен немедленно, второму придется ждать 100 мс и т. д. Последнему процессу, возможно, придется ждать целую секунду, если все остальные не блокируются за время кванта. Большинству пользователей секундная задержка вряд ли понравится.

Важен и тот фактор, что если установленное значение кванта больше среднего интервала работы процессора, переключение процессов будет происходить редко. Напротив, большинство процессов будут совершать блокирующую операцию прежде, чем истечет длительность кванта, вызывая переключение процессов. Устранение принудительных переключений процессов улучшает производительность системы, так как переключения процессов будут происходить только тогда, когда это логически необходимо, то есть когда процесс заблокировался и не может продолжать работу.

Вывод можно сформулировать следующим образом: слишком малый квант приведет к частому переключению процессов и небольшой эффективности, но слишком большой квант может привести к медленному реагированию на короткие интерактивные запросы. Значение кванта около 20–50 мс часто является разумным компромиссом.

Приоритетное планирование

В циклическом алгоритме планирования есть важное допущение о том, что все процессы равнозначны. В ситуации компьютера с большим числом пользователей это может быть не так. Например, в университете прежде всего должны обслуживаться деканы, затем профессора, секретари, уборщицы и лишь потом студенты. Необходимость принимать во внимание подобные внешние факторы приводит к **приоритетному планированию**. Основная идея проста: каждому процессу присваивается приоритет, и управление передается готовому к работе процессу с самым высоким приоритетом.

Даже на персональном компьютере с одним пользователем может происходить несколько процессов, отдельные из которых являются более важными, чем другие. Демон, отвечающий за пересылку электронной почты в фоновом режиме, имеет более низкий приоритет, чем процесс, отображающий на экране видеофильм в реальном времени.

Чтобы предотвратить бесконечную работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса с каждым тактом часов (то есть при каждом прерывании по таймеру). Если в результате приоритет текущего

процесса окажется ниже, чем приоритет следующего процесса, произойдет переключение. Возможно предоставление каждому процессу максимального отрезка времени работы. Как только время кончилось, управление передается следующему по приоритету процессу.

Приоритеты процессам могут присваиваться статически или динамически. На военной базе процессу, запущенному генералом, присваивается приоритет 100, полковником — 90, майором — 80, капитаном — 70, лейтенантом — 60 и т. д. А в коммерческом компьютерном центре выполнение заданий с высоким приоритетом может стоить 100 долларов в час, со средним — 75, с низким — 50. В системе UNIX есть команда *nice*, позволяющая пользователю добровольно снизить приоритет своих процессов, чтобы быть вежливым по отношению к остальным пользователям. Этой командой никто никогда не пользуется.

Система может динамически назначать приоритеты для достижения своих целей. Например, некоторые процессы сильно ограничены возможностями устройств ввода-вывода и большую часть времени проводят в ожидании завершения операций ввода-вывода. Когда бы ни потребовался процессор такому процессу, его следует немедленно предоставить, чтобы процесс мог начать следующий запрос ввода-вывода, который будет выполняться параллельно с вычислениями другого процесса. Если заставить процесс, ограниченный возможностями устройств ввода-вывода, длительное время ждать доступа к процессору, он будет неоправданно долго находиться в памяти. Простой алгоритм обслуживания процессов, ограниченных возможностями устройств ввода-вывода, состоит в установке приоритета, равного $1/f$, где f — часть использованного в последний раз кванта. Процесс, использовавший всего 1 мс из 50 мс кванта, получит приоритет 50, процесс, использовавший 25 мс, получит приоритет 2, а процесс, использовавший весь квант, получит приоритет 1.

Часто бывает удобно сгруппировать процессы в классы по приоритетам и использовать приоритетное планирование среди классов, но циклическое планирование внутри каждого класса. На рис. 2.24 представлена система с четырьмя классами приоритетов. Алгоритм планирования выглядит следующим образом: пока в классе 4 есть готовые к запуску процессы, они запускаются один за другим согласно алгоритму циклического планирования, и каждому отводится квант времени. При этом классы с более низким приоритетом не будут их беспокоить. Если в классе 4 нет готовых к запуску процессов, запускаются процессы класса 3 и т. д. Если приоритеты постоянны, до процессов класса 1 процессор может не дойти никогда.

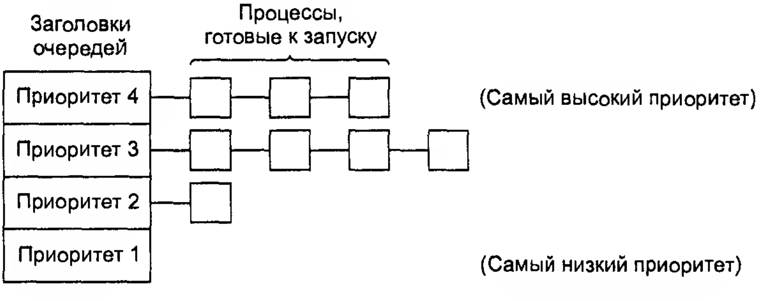


Рис. 2.24. Приоритетный алгоритм планирования с четырьмя классами приоритетов

Несколько очередей

Один из первых приоритетных планировщиков был реализован в системе CTSS (compatible time-shared system — совместимая система с разделением времени) [75]. Основной проблемой системы CTSS было слишком медленное переключение процессов, поскольку в памяти компьютера IBM 7094 мог находиться только один процесс. Каждое переключение означало выгрузку текущего процесса на диск и считывание нового процесса с диска. Разработчики CTSS быстро сообразили, что эффективность будет выше, если процессам, ограниченным возможностями процессора, выделять больший квант времени, чем если предоставлять им небольшие кванты, но часто. С одной стороны, это уменьшит количество перекачек из памяти на диск, а с другой — приведет к ухудшению времени отклика, как мы уже видели. В результате было разработано решение с классами приоритетов. Процессам класса с высшим приоритетом выделялся один квант, процессам следующего класса — два кванта, следующего — четыре кванта и т. д. Когда процесс использовал все отведенное ему время, он перемещался на класс ниже.

В качестве примера рассмотрим процесс, которому необходимо производить вычисления в течение 100 квантов. Вначале ему будет предоставлен один квант, затем он будет перекачан на диск. В следующий раз ему достанется 2 кванта, затем 4, 8, 16, 32, 64, хотя из 64 он использует только 37. В этом случае понадобится всего 7 перекачек (включая первоначальную загрузку) вместо 100, которые понадобились бы при использовании циклического алгоритма. Помимо этого, по мере погружения в очереди приоритетов процесс будет все реже запускаться, предоставляя процессор более коротким процессам.

Чтобы процессу, который при запуске считался долгим, но позже стал интерактивным, не погибнуть в недрах планирования, была разработана следующая стратегия. Как только с терминала приходит сигнал возврата каретки, процесс, соответствующий этому терминалу, переносится в класс высшего приоритета, поскольку предполагается, что он становится интерактивным. Однажды пользователь, запустивший процесс, сильно ограниченный возможностями процессора, обнаружил, что бездумное нажимание клавиши возврата каретки существенно уменьшает время отклика, и рассказал об этом друзьям. Мораль этой истории такова: осуществить задуманное на практике гораздо сложней, чем в теории.

Для разделения процессов по классам используются также многие другие алгоритмы. Например, в системе XDS 940, разработанной в Беркли [192], было четыре класса приоритетов, называвшихся: терминал, ввод-вывод, короткий квант и длинный квант. Когда запускался процесс, ожидающий вывода на терминал, он перемещался в класс высшего приоритета (терминал). Когда снималась блокировка процесса, ожидавшего доступа к диску, он перемещался во второй класс. Если к концу отведенного времени процесс все еще работал, он сначала перемещался в третий класс. Если процесс слишком много раз полностью использовал свой квант времени, не блокируясь на терминале или другом устройстве ввода-вывода, он перемещался в последний класс. Этот метод используется во многих системах для предоставления преимущества интерактивным процессам по сравнению с фоновыми.

«Самый короткий процесс — следующий»

Поскольку алгоритм «Кратчайшая задача — первая» минимизирует среднее оборотное время в системах пакетной обработки, хотелось бы использовать его и в инте-

рактивных системах. В известной степени это возможно. Интерактивные процессы чаще всего следуют схеме «ожидание команды, исполнение команды, ожидание команды, исполнение команды...» Если рассматривать выполнение каждой команды как отдельную задачу, можно минимизировать общее среднее время отклика, запуская первой самую короткую задачу. Единственная проблема состоит в том, чтобы понять, какой из ожидающих процессов самый короткий.

Один из методов основывается на оценке длины процесса, базирующейся на предыдущем поведении процесса. При этом запускается процесс, у которого оцененное время самое маленькое. Допустим, что предполагаемое время исполнения команды равно T_0 и предполагаемое время следующего запуска равно T_1. Можно улучшить оценку времени, взяв взвешенную сумму этих времен $aT_0 + (1 - a)T_1$. Выбирая соответствующее значение a, мы можем заставить алгоритм оценки быстро забывать о предыдущих запусках или, наоборот, помнить о них в течение долгого времени. Взяв $a = 1/2$, мы получим серию оценок:

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2.$$

После трех запусков вес T_0 в оценке уменьшится до 1/8.

Метод оценки следующего значения серии через взвешенное среднее предыдущего значения и предыдущей оценки часто называют **старением**. Этот метод применим во многих ситуациях, где необходима оценка по предыдущим значениям. Проще всего реализовать старение при $a = 1/2$. На каждом шаге нужно всего лишь добавить к текущей оценке новое значение и разделить сумму пополам (сдвинув вправо на 1 бит).

Гарантированное планирование

Принципиально другим подходом к планированию является предоставление пользователям реальных обещаний и затем их выполнение. Вот одно обещание, которое легко произнести и легко выполнить: если вместе с вами процессором пользуются n пользователей, вам будет предоставлено $1/n$ мощности процессора. И в системе с одним пользователем и n запущенными процессорами каждому достанется $1/n$ циклов процессора.

Чтобы выполнить это обещание, система должна отслеживать распределение процессора между процессами с момента создания каждого процесса. Затем система рассчитывает количество ресурсов процессора, на которое процесс имеет право, например время с момента создания, деленное на n. Теперь можно сосчитать отношение времени, предоставленного процессу, к времени, на которое он имеет право. Полученное значение 0,5 означает, что процессу выделили только половину положенного, а 2,0 означает, что процессу досталось в два раза больше, чем положено. Затем запускается процесс, у которого это отношение наименьшее, пока оно не станет больше, чем у его ближайшего соседа.

Лотерейное планирование

Хотя идея обещаний пользователям и их выполнения хороша, но ее трудно реализовать. Для более простой реализации предсказуемых результатов используется другой алгоритм, называемый **лотерейным планированием** [352].

В основе алгоритма лежит раздача процессам лотерейных билетов на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, «лотерея» может происходить 50 раз в секунду, и победитель получает 20 мс времени процессора.

Если перефразировать Джорджа Оруэлла: «Все процессы равны, но некоторые равнее других». Более важным процессам можно раздать дополнительные билеты, чтобы увеличить вероятность выигрыша. Если всего 100 билетов и 20 из них находятся у одного процесса, то ему достанется 20 % времени процессора. В отличие от приоритетного планировщика, в котором очень трудно оценить, что означает, скажем, приоритет 40, в лотерейном планировании все очевидно. Каждый процесс получит процент ресурсов, примерно равный проценту имеющихся у него билетов.

Лотерейное планирование характеризуется несколькими интересными свойствами. Например, если при создании процессу достается несколько билетов, то уже в следующей лотерее его шансы на выигрыш пропорциональны количеству билетов. Другими словами, лотерейное планирование обладает высокой отзывчивостью.

Взаимодействующие процессы могут при необходимости обмениваться билетами. Так, если клиентский процесс посылает сообщение серверному процессу и затем блокируется, он может передать все свои билеты серверному процессу, чтобы увеличить шанс запуска сервера. Когда серверный процесс заканчивает работу, он может вернуть все билеты обратно. Действительно, если клиентов нет, то серверу билеты вовсе не нужны.

Лотерейное планирование позволяет решать задачи, которые не решить с помощью других алгоритмов. В качестве примера можно привести видеосервер, на котором несколько процессов передают своим клиентам потоки видеоинформации с различной частотой кадров. Предположим, что процессы используют частоты 10, 20 и 25 кадров в секунду. Предоставив процессам соответственно 10, 20 и 25 билетов, можно реализовать загрузку процессора в желаемой пропорции 10:20:25.

Справедливое планирование

До сих пор мы предполагали, что каждый процесс управляется независимо от того, кто его хозяин. Поэтому если пользователь 1 создаст 9 процессов, а пользователь 2 — 1 процесс, то с использованием циклического планирования или в случае равных приоритетов пользователю 1 достанется 90 % процессора, а пользователю 2 всего 10.

Чтобы избежать подобных ситуаций, некоторые системы обращают внимание на хозяина процесса перед планированием. В такой модели каждому пользователю достается некоторая доля процессора, и планировщик выбирает процесс в соответствии с этим фактом. Если в нашем примере каждому из пользователей было обещано по 50 % процессора, то им достанется по 50 % процессора, независимо от количества процессов.

В качестве примера рассмотрим систему и двух пользователей, каждому из которых отведено по 50 % процессора. У первого пользователя четыре процесса: *A*, *B*, *C* и *D*, у второго один процесс *E*. Если используется циклическое планирование,

цепочка процессов, удовлетворяющая всем требованиям, будет выглядеть следующим образом:

A E B E C E D E A E B E C E D E...

С другой стороны, если первому пользователю положено вдвое больше ресурсов, чем второму, мы получим

A B E C D E A B E C D E...

Существует множество других решений, используемых в зависимости от конкретных представлений о справедливости.

Планирование в системах реального времени

В системах **реального времени** существенную роль играет время. Чаще всего одно или несколько внешних физических устройств генерируют входные сигналы, и компьютер должен адекватно на них реагировать в течение заданного промежутка времени. Например, компьютер в проигрывателе компакт-дисков получает биты от дисковода и должен за очень маленький промежуток времени конвертировать их в музыку. Если процесс конвертации будет слишком долгим, звук окажется искаженным. Подобные системы также используются для наблюдения за пациентами в палатах интенсивной терапии, в качестве автопилота самолета, для управления роботами на автоматизированном производстве. В любом из этих случаев запоздалая реакция ничуть не лучше, чем отсутствие реакции.

Системы реального времени делятся на **жесткие системы реального времени**, что означает наличие жестких сроков для каждой задачи (в них обязательно надо укладываться), и **гибкие системы реального времени**, в которых нарушения временного графика нежелательны, но допустимы. В обоих случаях реализуется разделение программы на несколько процессов, каждый из которых предсказуем. Эти процессы чаще всего бывают короткими и завершают свою работу в течение секунды. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Внешние события, на которые система должна реагировать, можно разделить на **периодические** (возникающие через регулярные интервалы времени) и **непериодические** (возникающие непредсказуемо). Возможно наличие нескольких периодических потоков событий, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события. Если в систему поступает m периодических событий, событие с номером i поступает с периодом P_i, и на его обработку уходит C_i секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1.$$

Системы реального времени, удовлетворяющие этому условию, называются **поддающимися планированию** или **планируемыми**.

В качестве примера рассмотрим гибкую систему реального времени с тремя периодическими сигналами с периодами в 100, 200 и 500 мс соответственно. Если на обработку этих сигналов уходит 50, 30 и 100 мс соответственно, система является поддающейся планированию, поскольку 0,5 + 0,15 + 0,2 < 1. Даже при добавлении четвертого сигнала с периодом в 1 с системой все равно можно будет управлять при помощи планирования, пока время обработки сигнала не будет превышать 150 мс. В этих расчетах существенным является предположение, что время переключения между процессами пренебрежимо мало.

Алгоритмы планирования для систем реального времени могут быть как статическими, так и динамическими. В первом случае все решения планирования принимаются заранее, еще до запуска системы. Во втором случае решения планирования принимаются по ходу дела. Статическое планирование применимо только при наличии достоверной информации о работе, которую необходимо выполнить, и о временном графике, которого нужно придерживаться. Динамическое планирование не нуждается в подобных ограничениях. На этом мы отложим изучение алгоритмов планирования и вернемся к ним в главе 7, посвященной мультимедийным системам реального времени.

Политика и механизм

Вплоть до этого момента мы подразумевали, что процессы в системе принадлежат различным пользователям и конкурируют за доступ к процессору. Чаще всего именно так все и выглядит, но возможна ситуация, в которой у одного процесса есть много дочерних процессов, работающих под его управлением. Например, у процесса, управляющего базой данных, может быть много дочерних процессов, обрабатывающих отдельные запросы или выполняющих конкретные функции (анализ запроса, доступ к диску и т. п.). Вполне возможно, что родительский процесс лучше представляет, какой из его дочерних процессов более важен (или для которого фактор времени более критичен), а какой — менее важен. К сожалению, ни один из рассмотренных нами планировщиков не может получать информацию от пользовательских процессов и учитывать ее при принятии решений планирования. В результате планировщики редко принимают оптимальное решение.

Решить проблему можно, разделив **механизм планирования** и **политику планирования**. Таким образом, мы реализуем ситуацию, в которой алгоритм планирования будет каким-либо образом параметризован, но параметры могут быть заданы процессом пользователя. Обратимся еще раз к примеру базы данных. Пусть ядро использует алгоритм приоритетного планирования, но существует системный запрос, посредством которого процесс может устанавливать (и менять) приоритеты своих дочерних процессов. В этом случае родительский процесс может управлять планированием дочерними процессами, хотя сам он планирования не осуществляет. Механизм определяется ядром, но политику задает процесс пользователя.

Планирование потоков

В случае нескольких процессов, каждый из которых разделен на несколько потоков, реализуются два уровня параллелизма: на уровне потоков и на уровне про-

цессов. Планирование в таких системах существенно зависит от того, поддерживаются ли потоки на уровне пользователя, на уровне ядра или и те и другие.

Для начала рассмотрим потоки на уровне пользователя. Поскольку ядро не знает о существовании потоков, оно выполняет обычное планирование, выбирая процесс *A* и предоставляя ему квант времени. Планировщик потоков внутри процесса *A* выбирает поток, например *A*1. Поскольку в случае потоков прерывания по таймеру нет, выбранный поток будет работать столько, сколько пожелает. Если он займет весь квант процесса *A*, ядро передаст управление другому процессу.

Когда управление снова перейдет к процессу *A*, поток *A*1 возобновит работу. Он будет продолжать потреблять все процессорное время, предоставляемое процессу *A*, пока не закончит свою работу. Впрочем, асоциальное поведение потока *A*1 на другие процессы не повлияет. Они будут продолжать получать долю процессорного времени, которую планировщик считает справедливой, независимо от того, что происходит внутри процесса *A*.

Теперь представим, что потокам процесса *A* нужно всего лишь 5 мс из отведенного кванта в 50 мс. Соответственно, каждый из них будет выполнять свою небольшую работу и возвращать процессор планировщику потоков. Это приведет к следующей цепочке: *A*1, *A*2, *A*3, *A*1, *A*2, *A*3, *A*1, *A*2, *A*3, *A*1, прежде чем управление будет передано процессу *B*. Эта ситуация представлена на рис. 2.25, *а*.

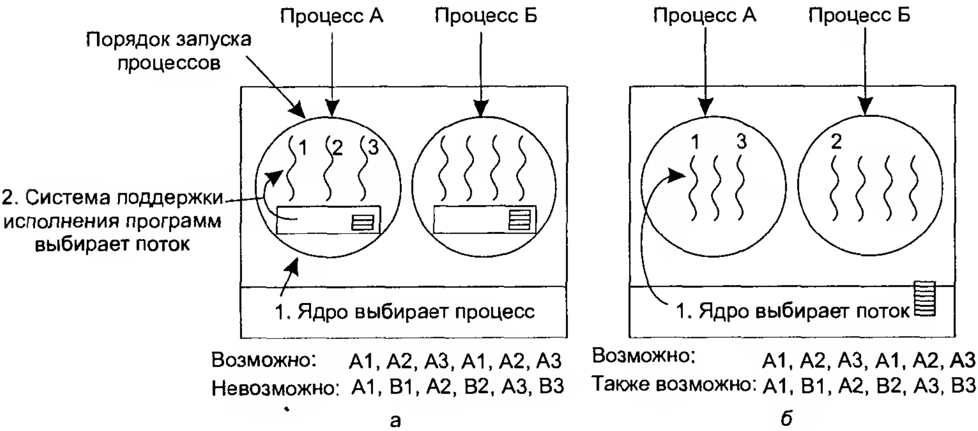


Рис. 2.25. Возможное планирование потоков: на уровне пользователя в случае кванта в 50 мс и потоков, блокирующихся через 5 мс (*а*); на уровне ядра с теми же характеристиками (*б*)

В качестве алгоритма планирования для системы поддержки исполнения программ можно взять любой из уже рассмотренных нами. Наиболее часто используются алгоритмы циклического и приоритетного планирования. Единственной проблемой является отсутствие таймера, который прерывал бы затянувшуюся работу потока.

Теперь рассмотрим потоки на уровне ядра. В этой ситуации ядро выбирает следующий поток. При этом ядро не обязано принимать во внимание, какой поток принадлежит какому процессу, хотя у него есть такая возможность. Потоку предоставляется квант времени и по истечении этого кванта управление передается

другому потоку. В случае кванта в 50 мс и потоков, блокирующихся через 5 мс, цепочка длиной в 30 мс может выглядеть так: *A*1, *B*1, *A*2, *B*2, *A*3, *B*3, что было невозможно в случае потоков на уровне пользователя. Эта ситуация представлена на рис. 2.25, *б*.

Основное различие между реализацией потоков на уровне пользователя и реализацией их на уровне ядра состоит в производительности. Для переключения потоков на уровне пользователя требуется выполнение всего нескольких машинных команд. Для переключения потоков на уровне ядра нужно выполнить полное переключение контекста с заменой карты памяти и аннулированием кэша, что выполняется на несколько порядков медленнее. С другой стороны, при реализации потоков на уровне пользователя блокировка потока на устройстве ввода-вывода блокирует весь процесс, чего не случается с потоками на уровне ядра.

Поскольку ядро знает, что на переключение от потока процесса *A* к потоку процесса *B* будет затрачено больше ресурсов, чем на передачу управления следующему потоку процесса *A* (из-за карты памяти и кэша), эта информация может учитываться при принятии решения планирования. Например, при наличии двух одинаково важных потоков, один из которых принадлежит тому же процессу, что и только что блокированный поток, а второй — другому процессу, предпочтение будет отдано первому потоку.

Еще одним важным фактором является возможность совместного использования потоков на уровне пользователя и специализированного планировщика потоков. Рассмотрим, например, web-сервер на рис. 2.7. Пусть один рабочий поток только что заблокирован, а диспетчер и два оставшихся рабочих потока находятся в состоянии готовности. Который из них будет запущен? Система поддержки исполнения программ, которая обладает информацией о задаче каждого потока, выберет следующим диспетчера, чтобы он запустил следующий рабочий поток. Подобная стратегия увеличивает степень параллелизма в среде, где рабочие потоки часто блокируются на обращениях к диску. В случае потоков на уровне ядра оно не знает, чем занимается каждый поток (хотя у них могут быть разные приоритеты). В целом специализированные планировщики потоков лучше управляют приложениями, чем ядро.

Изучение процессов и потоков

В главе 1 мы рассмотрели некоторые текущие исследования структуры операционных систем. В этой и следующих главах мы рассмотрим более конкретные исследования и начнем с процессов. Как вы поймете позже, что-то уже устоялось, что-то еще нет. Большая часть исследований направлена на новые темы.

Концепция процесса является хорошим примером уже разработанной темы. Практически в каждой системе есть представление о процессе как о контейнере для группировки связанных ресурсов, таких как адресное пространство, потоки, открытые файлы, доступ к защищенным ресурсам и т. п. Способ группировки слегка различается в разных системах, но эти различия видны лишь на уровне проектирования. С основной идеей уже почти никто не спорит, и новых исследований по этой теме немного.

Идея потоков более свежая, поэтому по этой теме до сих пор ведется несколько исследований. Хаузер (Hauser) [149] рассматривал, как реальные программы используют потоки, и выработал 10 различных парадигм использования потоков. Планирование потоков (с одним и несколькими процессорами) до сих пор изучается, и эта тема близка многим исследователям [36, 47, 59, 73, 101, 120, 264]. Тем не менее не так уж много разработчиков операционных систем ломают головы день и ночь, ощущая катастрофический недостаток алгоритмов планирования. Похоже, что это поле исследований пока не ощущает инфляции спроса.

Тесно связаны с потоками темы синхронизации потоков и взаимного исключения. В 70-х и 80-х годах этими исследованиями занимались все, поэтому в настоящее время работ по этим темам немного. Проводимые исследования в основном направлены на повышение эффективности (например, [207]), разработку методов детектирования ошибок синхронизации [293] и переработку старых концепций [319]. Наконец, до сих пор производятся POSIX-совместимые пакеты потоков [9, 235].

Резюме

При рассмотрении операционных систем с точки зрения концептуальной модели, состоящей из последовательных процессов, работающих параллельно, эффект прерываний скрыт. Процессы можно динамично создавать и завершать. У каждого процесса есть собственное адресное пространство.

Для некоторых приложений удобно иметь несколько потоков управления в одном процессе. Эти потоки планируются независимо друг от друга и у каждого есть собственный стек, но все потоки одного процесса совместно используют общее адресное пространство. Потоки можно реализовать в пространстве пользователя или в пространстве ядра.

Процессы могут взаимодействовать между собой с помощью примитивов межпроцессного взаимодействия, таких как семафоры, мониторы или сообщения. Эти примитивы используются для исключения ситуации, в которой два процесса одновременно находятся в своих критических областях, что приводило бы к хаосу. Процесс может находиться в состоянии действия, готовности или блокировки и может менять состояние в случае выполнения им (или другим процессом) примитива межпроцессного взаимодействия. Примерно так же выполняется взаимодействие потоков.

Примитивы межпроцессного взаимодействия используются для решения таких проблем, как проблема производителя и потребителя, проблема обедающих философов, проблема читателей и писателей и проблема спящего брадобрея. Но даже при использовании примитивов необходимо отслеживать ситуации, приводящие к ошибкам и взаимоблокировкам.

Известны многие алгоритмы планирования. Некоторые из них, такие как «Кратчайшая задача — первая», используются в основном в системах пакетной обработки данных. Другие используются в интерактивных системах (циклическое планирование, приоритетное планирование, многоуровневые очереди, гарантированное планирование, лотерейное планирование и разделенное планирование). В некоторых системах различаются механизм планирования и политика планирования, что позволяет пользователю управлять алгоритмом планирования.

Вопросы

1. На рис. 2.2 представлены три состояния процессов. Теоретически между этими тремя состояниями могут быть шесть переходов, но на рисунке показано только четыре. Возможна ли ситуация, в которой произойдет один или оба пропущенных перехода?
2. Представьте, что вы разрабатываете архитектуру компьютера высокого уровня, который переключает процессы аппаратно, а не с помощью прерываний. Какая информация потребуется процессору? Опишите возможную реализацию переключения процессов с использованием аппаратного обеспечения.
3. На всех существующих компьютерах как минимум часть обработчиков прерываний написана на ассемблере. Почему?
4. Когда прерывание или системный запрос передает управление операционной системе, обычно используется область стека ядра, а не стек прерванного процесса. Почему?
5. В тексте утверждалось, что модель, представленная на рис. 2.4, *а*, не подходит для файлового сервера с кэшем в памяти. Почему? Может ли каждый процесс иметь собственный кэш?
6. В табл. 2.3 набор регистров относится к элементам потока, а не процесса. Почему? В конце концов, в компьютере всего один набор регистров.
7. В случае разветвления многопоточного процесса возникнет проблема, если дочернему процессу достанутся копии всех потоков. Пусть один из исходных потоков ожидал ввода с клавиатуры. После ветвления два потока будут ожидать ввода с клавиатуры, по одному в каждом процессе. Может ли такая проблема возникнуть в случае ветвления однопоточного процесса?
8. На рис. 2.7 представлен многопоточный web-сервер. Если единственным способом прочитать информацию из файла является обычный блокирующий системный запрос read, что, по вашему мнению, используется для сервера — потоки на уровне пользователя или на уровне ядра? Почему?
9. Почему поток должен добровольно отказываться от доступа к процессору, вызывая *thread_yeld*? В конце концов, поток может никогда не получить процессор обратно, поскольку прерывания по таймеру нет.
10. Может ли поток быть прерван прерыванием по таймеру? Если да, то при каких обстоятельствах? Если нет, то почему?
11. В этой задаче вам нужно сравнить считывание файла через однопоточный и многопоточный файловые серверы. Получение запроса, его диспетчеризация и обработка занимают 15 мс при условии наличия требуемых данных в блочном кэше. В каждом третьем случае требуется обращение к диску, занимающее 75 мс, в течение которых поток находится в состоянии ожидания. Сколько запросов в секунду обработает однопоточный сервер? А многопоточный?
12. В тексте был описан многопоточный web-сервер и показано, почему он лучше, чем однопоточный сервер или конечный автомат. Возможна ли ситуация, в которой однопоточный сервер будет лучше? Приведите пример.

13. В разделе, посвященном глобальным переменным в потоках, процедура *create_global* использовалась для выделения участка памяти для хранения указателя на переменную, но не самой переменной. Является это условие необходимым, или процедуры будут точно так же работать, если в этом участке памяти будут храниться сами переменные?
14. Рассмотрим систему, в которой все потоки реализованы в пространстве пользователя и система поддержки исполнения программ получает прерывание по таймеру раз в секунду. Пусть прерывание получено в тот момент, когда поток контролируется системой управления программ. К какой проблеме это может привести? Предложите способ решения этой проблемы.
15. Рассмотрим операционную систему, в которой нет никакого механизма, похожего на системный запрос select, чтобы узнать заранее, приведет ли к блокировке считывание информации из файла, канала или устройства. Зато в этой системе есть аварийный таймер, прерывающий блокированные системные запросы. Возможна ли в такой системе реализация потоков в пространстве пользователя? Поясните.
16. Может ли в случае потоков в пространстве пользователя возникнуть проблема инверсии приоритета, рассмотренная в разделе «Примитивы межпроцессного взаимодействия»? Почему?
17. В системе с потоками на уровне пользователя каждому потоку соответствует собственный стек или каждому процессу? А в системе с потоками на уровне ядра? Поясните.
18. Что такое состояние состязания?
19. При разработке компьютера он сначала моделируется программой, выполняющей одновременно только одну команду. Даже компьютер с несколькими процессорами моделируется также последовательно. Может ли в такой ситуации возникнуть состояние соревнования, когда одновременных событий нет?
20. Будет ли работать решение активного ожидания с использованием переменной *turn* (см. рис. 2.16) в случае двух процессоров, совместно использующих общую память?
21. Будет ли решение Петерсона проблемы взаимного исключения (листинг 2.1) работать в случае планирования процессов с переключениями? В случае планирования без переключений?
22. Рассмотрим компьютер, в котором нет инструкции TSL, но есть инструкция для обмена содержимого регистра и слова памяти за одну неделимую операцию. Можно ли использовать эту инструкцию для написания программы *enter_region*, аналогичной листингу 2.3?
23. Опишите коротко, как реализовать семафоры в операционной системе, умеющей блокировать прерывания.
24. Покажите, как можно реализовать считающие семафоры (то есть семафоры, способные хранить произвольные значения) при помощи только бинарных семафоров и обычных машинных команд.

25. Если в системе всего два процесса, имеет ли смысл использовать для их синхронизации барьер? Почему?
26. В разделе «Примитивы межпроцессного взаимодействия» была описана ситуация с высокоприоритетным процессом *H* и низкоприоритетным процессом *L*, которая приводила к вечному зацикливанию процесса *H*. Может ли возникнуть подобная проблема, если вместо приоритетного планирования использовать циклическое? Поясните.
27. Можно ли синхронизировать два потока одного процесса, используя семафор в ядре, если потоки реализованы в пространстве ядра? Если потоки реализованы в пространстве пользователя? Предполагается, что потоки остальных процессов не имеют доступа к семафору. Поясните ответы.
28. Синхронизация в мониторах происходит с использованием **переменных состояния** и двух специальных операций, `wait` и `signal`. Более общая форма синхронизации предполагает один примитив `waituntil` с произвольным булевым предикатом в качестве параметра. Например,

    ```
    waituntil x<0 or y+z<n
    ```

    В этом случае примитив `signal` больше не нужен. Эта схема существенно более общая, чем схема Хоара и Бринча Хансена, но тем не менее она не используется. Почему? *Подсказка*: подумайте о реализации.
29. В ресторане быстрого обслуживания есть четыре категории обслуживающего персонала: 1) работники, принимающие заказы; 2) повара, готовящие пищу; 3) специалисты по упаковке блюд и 4) кассиры, принимающие у клиентов деньги и выдающие еду. Каждому из видов персонала можно сопоставить последовательный процесс взаимодействия. Какой формой межпроцессного взаимодействия они пользуются? Свяжите эту модель с процессами в UNIX.
30. Рассмотрим систему передачи сообщений, использующую почтовые ящики. При попытке послать сообщение в полный ящик или получить сообщение из пустого ящика процесс не блокируется, а получает код ошибки. Затем процесс повторяет попытку, пока она не окажется успешной. Приведет ли подобная схема к условию состязания?
31. Почему в процедуре *take_forks* решения задачи обедающих философов (листинг 2.11) переменной состояния присваивается значение *HUNGRY*?
32. Рассмотрим процедуру *put_forks* в листинге 2.11. Пусть переменной состояния присваивается значение *THINKING* после двух вызовов процедуры *test*, а не до. Как это повлияет на решение?
33. Проблему читателей и писателей можно формулировать по-разному, в зависимости от того, какие процессы и в какое время могут быть запущены. Тщательно опишите три варианта задачи, в каждом из которых предоставляется (или не предоставляется) преимущество одной из категорий. В каждом варианте укажите, что происходит, когда читающий или пишущий процесс готов обратиться к базе данных, и что происходит, когда процесс заканчивает работу с базой.

34. Компьютеры CDC 6600 могут обрабатывать до 10 процессов ввода-вывода одновременно, используя интересную форму циклического планирования, называемую **разделением процессора**. Переключение между процессами происходит после каждой команды, поэтому команда 1 поступает от первого процесса, команда 2 — от второго и т. д. Переключение процессов производится аппаратными средствами, и издержки равны нулю. Если в отсутствие других процессов процессу для выполнения работы нужно T секунд, сколько ему потребуется времени в случае n процессов?
35. Обычно планировщики с циклическим алгоритмом поддерживают список процессов, готовых к работе, причем каждый процесс находится в списке в единственном экземпляре. Что произойдет, если процесс окажется в списке дважды? Существует ли причина, по которой подобное изменение будет полезным?
36. Можно ли определить путем анализа исходного кода принадлежность процесса к процессам, ограниченным возможностями процессора или устройств ввода-вывода? Как это можно определить во время выполнения программы?
37. В разделе «Когда планировать?» упоминалось, что планирование может быть усовершенствовано, если важный процесс может сыграть свою роль в выборе процесса, следующего за ним. Опишите ситуацию, в которой это можно использовать, и объясните, как.
38. Измерения показали, что время выполнения среднестатистического процесса до блокировки ввода-вывода равно T. На переключение между процессами уходит время S, которое теряется впустую. Напишите формулу расчета эффективности для циклического планирования с квантом Q, принимающим следующие значения:
    а) $Q = \infty$,
    б) $Q > T$,
    в) $S < Q < T$,
    г) $Q = S$,
    д) Q около 0.
39. Запуска ожидают пять задач. Предполагаемое время выполнения задач составляет 9, 6, 3, 5 и X. В каком порядке их следует запустить, чтобы минимизировать среднее время отклика? (Ответ должен зависеть от X.)
40. Пять пакетных задач, A, B, C, D, E поступают в компьютерный центр практически одновременно. Ожидается, что время их выполнения составит 10, 6, 2, 4 и 8 мин. Их установленные приоритеты составляют 3, 5, 2, 1 и 4, причем 5 — высший приоритет. Определите среднее оборотное время для каждого из следующих алгоритмов планирования, пренебрегая временем, теряющимся при переключении между процессами:
    а) циклическое планирование;
    б) приоритетное планирование;
    в) «первым пришел — первым обслужен» (в порядке 10, 6, 2, 4, 8);
    г) «кратчайшая задача — первая».

В случае *а* предполагается, что система многозадачная и каждой задаче достается справедливая доля процессорного времени. В случаях *б—г* предполагается, что в каждый момент времени запущена одна задача, работающая вплоть до завершения. Все задачи ограничены исключительно возможностями процессора.

41. Процессу, запущенному в системе CTSS, для завершения необходимо 30 квантов. Сколько раз он будет перекачан на диск, считая самый первый раз (прежде, чем он был запущен)?

42. Предложите защиту от обмана системы приоритетов CTSS случайными нажатиями клавиши возврата каретки.

43. Для предсказания времени выполнения используется алгоритм старения с $a = 1/2$. Предыдущие четыре значения времени составляли 40, 20, 40 и 15 мс (первое значение — самое давнее). Оцените следующее время выполнения.

44. В гибкую систему реального времени поступает четыре периодических сигнала с периодами 50, 100, 200 и 250 мс. На обработку каждого сигнала требуется 35, 20, 10 и x мс времени процессора. Укажите максимальное значение x, при котором система остается поддающейся планированию.

45. Объясните причины широкого использования двухуровневого планирования.

46. Рассмотрите систему, в которой желательно разделить механизм и стратегию планирования потоков на уровне ядра. Предложите средства достижения этой цели.

47. Напишите сценарий оболочки, которая создает файл, содержащий последовательные числа, путем считывания последнего числа, прибавления к нему единицы и записывания результата в конец файла. Запустите одну копию сценария в качестве фонового процесса и одну — в качестве приоритетного процесса. Сколько времени пройдет, прежде чем возникнет состояние соревнования? Что в данной модели является критической областью? Измените сценарий, чтобы избежать состояния состязания. (*Подсказка*: воспользуйтесь следующим выражением, чтобы заблокировать файл данных.)

```
In file file.lock
```

48. Предположим, что у вас есть операционная система с семафорами. Реализуйте систему сообщений. Напишите процедуры для отсылки и получения писем.

49. Решите задачу обедающих философов, используя мониторы вместо семафоров.

50. Некий университет решил продемонстрировать свою политкорректность, применив известную доктрину верховного суда США «Равенство порознь равенством не является» не только к цвету кожи, но и к полу. Результатом этого решения явились совместные ванные комнаты в общежитиях. Тем не менее в поддержку исторически сложившейся традиции университет постановляет, что если в ванной комнате есть женщина, то другая женщина

может туда зайти, а мужчина не может, и наоборот. На двери ванной есть индикатор, показывающий, в каком из трех состояний находится ванная:

а) никого нет;

б) в ванной женщины;

в) в ванной мужчины.

Напишите на своем любимом языке программирования следующие процедуры: *woman_wants_to_enter*, *man_wants_to_enter*, *woman_leaves*, *man_leaves*. Вы можете использовать любые счетчики и методы синхронизации.

51. Перепишите программу из листинга 2.12, чтобы она обрабатывала большее количество процессов.
52. Напишите алгоритм для решения задачи производителя и потребителя с использованием потоков и разделенного буфера. Однако не используйте семафоры и другие примитивы синхронизации для защиты совместно используемых структур данных, разрешите каждому потоку доступ по мере необходимости. Воспользуйтесь для обработки условий полного и пустого буфера функциями `sleep` и `wakeup`. Посмотрите, сколько времени пройдет до появления неустранимого состояния состязания. Например, производитель может периодически выдавать на печать число, но не чаще, чем раз в минуту, поскольку процесс ввода-вывода может повлиять на состояние состязания.
53. Для получения большего приоритета процесс можно поместить в очередь кругового алгоритма планирования больше одного раза. Этого же эффекта можно достигнуть, запустив несколько копий программы, каждая из которых будет обрабатывать свою часть пула данных. Для начала напишите программу, которая проверяет простоту чисел из списка. Затем разработайте способ разрешить одновременный запуск нескольких копий программы, причем так, чтобы две копии программы не обращались к одному числу. Удастся ли вам ускорить обработку списка, запустив несколько копий программы? Имейте в виду, что результат будет зависеть от того, чем еще занят ваш компьютер: при отсутствии других процессов улучшения результатов не будет, но если параллельно решаются другие задачи, несколько копий программы позволят получить больший процент времени процессора.

Глава 3
Взаимоблокировка

В компьютерных системах существует большое количество ресурсов, каждый из которых в конкретный момент времени может использоваться только одним процессом. В качестве таких примеров можно привести принтеры, накопители на магнитной ленте и элементы внутренних таблиц системы. Появление двух процессов, одновременно передающих данные на принтер, приведет к печати бессмысленного набора символов. Наличие двух процессов, использующих один и тот же элемент таблицы файловой системы, обязательно станет причиной разрушения файловой системы. Поэтому все операционные системы обладают способностью предоставлять процессу эксклюзивный доступ (по крайней мере, временный) к определенным ресурсам.

Часто для выполнения прикладных задач процесс нуждается в исключительном доступе не к одному, а к нескольким ресурсам. Предположим, например, что каждый из двух процессов хочет записать отсканированный документ на компакт-диск. Процесс *A* запрашивает разрешение на использование сканера и получает его. Процесс *B* запрограммирован по-другому, поэтому сначала запрашивает устройство для записи компакт-дисков и также получает его. Затем процесс *A* обращается к устройству для записи компакт-дисков, но запрос отклоняется до тех пор, пока это устройство занято процессом *B*. К сожалению, вместо того чтобы освободить устройство для записи компакт-дисков, *B* запрашивает сканер. В этот момент процессы заблокированы и будут вечно оставаться в этом состоянии. Такая ситуация называется **тупиком**, **тупиковой ситуацией** или **взаимоблокировкой**.

Взаимоблокировки могут возникать между различными машинами. Во многих офисах есть локальная сеть, включающая в себя множество компьютеров. Часто такие устройства, как сканеры, устройства для записи компакт-дисков, принтеры и накопители на магнитной ленте, присоединены к сети как ресурсы совместного доступа, то есть доступны любому пользователю на любой машине. Если эти ресурсы позволяется резервировать удаленно (то есть с домашней машины пользователя), может возникнуть аналогичный описанному выше вид тупиковых ситуаций. Более сложные ситуации могут стать причиной тупиков, вовлекающих три, четыре и более устройств и пользователей.

Взаимоблокировки могут произойти во множестве других ситуаций помимо запросов выделенных устройств ввода-вывода. В системах баз данных программа может оказаться вынужденной заблокировать несколько записей, чтобы избежать состояния конкуренции. Если процесс *A* блокирует запись *R*1, процесс *B* блокирует запись *R*2, а затем каждый процесс попытается заблокировать чужую запись,

мы также окажемся в тупике. Таким образом, взаимоблокировки появляются при работе как с аппаратными, так и с программными ресурсами.

В этой главе мы рассмотрим тупиковые ситуации более подробно, увидим, как они возникают, и изучим некоторые способы, позволяющие предупредить тупиковые ситуации или избежать их. Хотя информация о тупиках представлена в контексте операционной системы, они также могут встретиться в системах баз данных и во множестве других ситуаций, поэтому этот материал на самом деле применим к широкому ряду систем, работающих с несколькими процессами. О взаимоблокировках было написано много книг и статей. Библиографии по теме дважды публиковались в *Operating Systems Review* и к ним следует обращаться за справочной информацией ([246, 370]). Несмотря на то что эти библиографии написаны давно, большая часть работ по взаимоблокировкам была проделана до 1980 года, поэтому данные книги полезны до сих пор.

Ресурсы

Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к устройствам, файлам и т. д. Чтобы максимально обобщить рассказ о взаимоблокировках, мы будем называть объекты предоставления доступа **ресурсами**. Ресурсом может быть аппаратное устройство (например, накопитель на магнитной ленте) или часть информации (например, закрытая запись в базе данных). В компьютере существует масса различных ресурсов, к которым могут происходить обращения. Кроме того, в системе может оказаться несколько идентичных экземпляров какого-либо ресурса, например три накопителя на магнитных лентах. Если в системе есть несколько экземпляров ресурса, то в ответ на обращение к нему может предоставляться любая из доступных копий. Короче говоря, ресурс — это все то, что может использоваться только одним процессом в любой момент времени.

Выгружаемые и невыгружаемые ресурсы

Ресурсы бывают двух типов: выгружаемые и невыгружаемые. **Выгружаемый ресурс** можно безболезненно забирать у владеющего им процесса. Образцом такого ресурса является память. Рассмотрим, например, систему с пользовательской памятью размером 32 Мбайт, одним принтером и двумя процессами по 32 Мбайт, каждый из которых хочет что-то напечатать. Процесс *A* запрашивает и получает принтер, затем начинает вычислять данные для печати. Еще не закончив расчеты, он превышает свой квант времени и выгружается на диск в область подкачки.

Теперь работает процесс *B* и безуспешно пытается обратиться к принтеру. В данный момент мы получили потенциальную тупиковую ситуацию, потому что процесс *A* использует принтер, а процесс *B* занимает память, и ни один из них не может продолжать работу без ресурса, удерживаемого другим. К счастью, можно выгрузить (забрать) память у процесса *B*, переместив его на диск в область подкачки и скачав с диска в память процесс *A*. Теперь процесс *A* может закончить вычисления, выполнить печать и затем освободить принтер. Взаимоблокировки не происходит.

Невыгружаемый ресурс, в противоположность выгружаемому, — это такой ресурс, который нельзя забрать от текущего владельца, не уничтожив результаты вычислений. Если в момент записи компакт-диска внезапно отнять у процесса устройство для записи и передать его другому процессу, то в результате мы получим испорченный компакт-диск. Устройство для записи компакт-дисков не является выгружаемым в произвольный момент времени ресурсом.

Вообще говоря, взаимоблокировки касаются невыгружаемых ресурсов. Потенциальные тупиковые ситуации, в которые вовлечен противоположный вид ресурсов, обычно можно разрешить с помощью перераспределения ресурсов от одного процесса к другому. Поэтому мы сконцентрируем свое внимание на невыгружаемых ресурсах.

Последовательность событий, необходимых для использования ресурса, представлена ниже в абстрактной форме.

1. Запрос ресурса.
2. Использование ресурса.
3. Возврат ресурса.

Если ресурс недоступен, когда он требуется, то запрашивающий его процесс вынужден ждать. В некоторых операционных системах при неудачном обращении к ресурсу процесс автоматически блокируется и возобновляется только после того, как ресурс становится доступным. В других системах запрос ресурса, получивший отказ, возвращает код ошибки, тогда вызывающий процесс может подождать немного и повторить попытку заново.

Процесс, чье обращение к ресурсу оказалось неудачным, обычно дальше попадает в короткий цикл: запрос ресурса, затем режим ожидания, потом очередная попытка. Хотя этот процесс не блокирован, он во всех смыслах ведет себя, как заблокированный, поскольку не может выполнить никакой полезной работы. В дальнейшем мы будем предполагать, что когда процессу отказывается в предоставлении ресурса, он переходит в режим ожидания.

Истинная природа запросов ресурсов сильно зависит от системы. В некоторых системах существует системный вызов request, позволяющий процессам запрашивать ресурсы явно. В других случаях единственный вид ресурсов, известных операционной системе, — это специальные файлы, которые в каждый данный момент времени могут открыть только один процесс. Они открываются с помощью обычного вызова open. Если файл уже используется, вызывающая программа блокируется до тех пор, пока текущий владелец файла не закроет его.

Получение ресурса

Для некоторых видов ресурсов, таких как записи в базе данных, управление использованием ресурсов зависит от самих пользовательских процессов. Один из способов, делающих возможным пользовательское управление, заключается в присоединении семафора к каждому из ресурсов. Все семафоры в исходном состоянии равны 1. С тем же успехом можно использовать мьютексы. Три шага, перечисленные выше, выполняются следующим образом: сначала для запроса ресурса исполь-

зуется вызов down, примененный к семафору, затем программа использует ресурс и, наконец, используется вызов up для его освобождения. Эти шаги представлены в листинге 3.1, *а*.

Листинг 3.1. Использование семафора для защиты ресурсов: один ресурс (*а*); два ресурса (*б*)

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
        down(&resource_1);
        use_resource_1();
        up(&resource_1);
}
```

а

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources();
        up(&resource_2);
        up(&resource_1);
}
```

б

Иногда процессы нуждаются в двух и более ресурсах. Их можно получать последовательно, как показано в листинге 3.1, *б*. Если требуется больше двух ресурсов, их запрашивают непосредственно один за другим.

Пока все хорошо. Эта схема работает прекрасно до тех пор, пока она касается только одного процесса. Конечно, при наличии всего лишь одного процесса отсутствует необходимость формального приобретения ресурсов, поскольку не возникает соперничества за их использование.

Теперь рассмотрим ситуацию с двумя процессами *A* и *B* и двумя ресурсами. В листинге 3.2 показаны два сценария: *а* — оба процесса получают ресурсы в одном и том же порядке; *б* — они запрашивают ресурсы в разном порядке. Разница может показаться несущественной, но это не так.

В листинге 3.2, *а* один из процессов запрашивает первый ресурс, опережая второй процесс. Затем этот же процесс успешно получает второй ресурс и выполняет свою работу. Если второй процесс попытается получить ресурс 1 до его освобождения, второй процесс просто будет заблокирован до тех пор, пока не станет доступен ресурс.

Другая ситуация представлена в листинге 3.2, *б*. Может случиться так, что один из процессов получит оба ресурса и эффективно блокирует другой процесс до завершения своей работы. Однако также может произойти и то, что процесс *A* займет ресурс 1, а процесс *B* получит ресурс 2. Теперь, когда они попытаются запросить еще по одному ресурсу, каждый из них будет заблокирован. Ни один из двух процессов не сможет когда-либо заработать снова. Это ситуация взаимоблокировки.

Здесь мы видим, как проявляется несущественное различие в коде программы — последовательность запросов ресурсов, — которое вызывает разницу между работающей программой и программой, завершающейся аварийно, причем с трудно обнаруживаемой причиной ошибки. Поскольку взаимоблокировки могут происходить так просто, на поиски методов борьбы с ними было направлено большое количество исследований. В этой главе подробно будут обсуждены тупиковые ситуации и то, что можно в таких ситуациях предпринять.

Листинг 3.2. Код, не приводящий к тупику (а); код с потенциальной взаимоблокировкой (*б*)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;
void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}
void process_B(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}
```

а

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;
void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1)f
}
void process_B(void) {
      down(&resource_2);
      down(&resource_1);
      use_both_resources();
      up(&resource_1);
      up(&resource_2);
}
```

б

Введение

Взаимоблокировки или тупиковые ситуации формально можно определить так:

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активировать любой другой процесс в группе, и все процессы продолжают ждать до бесконечности. В этой модели мы предполагаем, что процессы имеют только один поток и что нет прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе.

В большинстве случаев событием, которого ждет каждый процесс, является возврат какого-либо ресурса, в данный момент занятого другим участником группы. Другими словами, каждый участник в группе процессов, зашедших в тупик, ждет доступа к ресурсу, принадлежащему заблокированному процессу. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться. Количество процессов и количество и вид ресурсов, имеющихся и запрашиваемых, здесь не важны. Результат остается тем же самым для любого вида ресурсов, аппаратных и программных.

Условия взаимоблокировки

В [70] Коффман (Coffman) и другие исследователи доказали, что для возникновения ситуации взаимоблокировки должны выполняться четыре условия:

1. Условие взаимного исключения. Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.

2. Условие удержания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурса. У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для того чтобы произошла взаимоблокировка, должны выполниться все эти четыре условия. Если хоть одно из них отсутствует, тупиковая ситуация невозможна.

Следует отметить, что каждое условие относится к стратегии определения того, что в системе позволяется, а что — нет. Может ли данный ресурс быть отдан одновременно больше чем одному процессу? Может ли процесс удерживать один ресурс и запрашивать второй? Можно ли отнять ресурс у процесса? Может ли существовать циклическое ожидание? Позже мы увидим, как можно разрушать взаимоблокировки с помощью сведения «на нет» некоторых из этих условий.

Моделирование взаимоблокировок

В [156] Холт (Holt) показал, как можно смоделировать четыре условия возникновения тупиков, используя направленные графы. Графы имеют два вида узлов: процессы, показанные кружочками, и ресурсы, нарисованные квадратиками. Ребро, направленное от узла ресурса (квадрат) к узлу процесса (круг), означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом. На рис. 3.1, *а* ресурс *R* в настоящее время отдан процессу *A*.

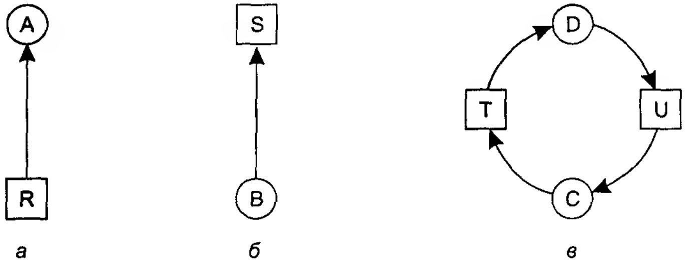


Рис. 3.1. Графы распределения ресурсов: ресурс занят (*а*); запрос ресурса (*б*); взаимоблокировка (*в*)

Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу. На рис. 3.1, *б* процесс *B* ждет ресурс *S*. На рис. 3.1, *в* мы видим взаимоблокировку: процесс *C* ожидает ресурс *T*, удерживаемый в настоящее время процессом *D*. Процесс *D* вовсе не намеревается освобождать ресурс *T*, потому что он ждет ресурс *U*, используемый процессом *C*. Оба процесса будут ждать до бесконечности.

Цикл в графе означает наличие взаимоблокировки, циклично включающей процессы и ресурсы (предполагается, что в системе есть по одному ресурсу каждого вида). В этом примере циклом является последовательность *C-T-D-U-C*.

Теперь рассмотрим пример того, как можно использовать графы ресурсов. Представим, что у нас есть три процесса: *A*, *B* и C, и три ресурса: *R*, *S* и *T*. Последовательность запросов и возвратов ресурсов для трех процессов показаны на рис. 3.2, *а—в*. Операционная система может запустить любой незаблокированный процесс в любой момент времени, значит, она может решить запустить сначала процесс *A*. Процесс *A* будет выполняться до тех пор, пока не закончит всю свою работу, затем будет запущен процесс *B* до его завершения и, наконец, процесс *C*.

Такой порядок не приводит к взаимоблокировке (потому что при нем не возникает соперничества за использование ресурсов), но при нем также вообще нет параллельной работы. Кроме запросов и возвратов ресурсов, процессы выполняют вычисления и ввод-вывод данных. Когда процессы работают последовательно, невозможна ситуация, при которой один процесс использует процессор, в то время как другой ждет завершения операции ввода-вывода. Таким образом, строго последовательная работа процессов не может быть оптимальной. С другой стороны, если вообще ни один процесс не выполняет операций ввода-вывода, алгоритм «кратчайшая задача — первая» работает лучше, чем циклический, поэтому в некоторой обстановке последовательный запуск всех процессов может быть наилучшим.

Теперь предположим, что процессы выполняют как расчеты, так и ввод-вывод, так что циклический алгоритм планирования является рациональным. Запросы ресурсов могут происходить в порядке, указанном на рис. 3.2, *г*. Если эти шесть запросов будут осуществлены в такой последовательности, в результате мы получим шесть графов, показанных на рис. 3.2, *д—к*. После запроса 4 процесс *A* блокируется в ожидании ресурса *S* (рис. 3.2, *з*). На двух следующих шагах также блокируются процессы *B* и *C*, в конечном счете приводя к циклу и взаимоблокировке на рис. 3.2, *к*.

Однако, как мы упоминали ранее, операционная система не обязана запускать процессы в каком-то особом порядке. В частности, если выполнение отдельного запроса приводит в тупик, операционная система может просто приостановить процесс без удовлетворения запроса (то есть не выполняя план процесса) до тех пор, пока это безопасно. На рис. 3.2 операционная система могла бы приостановить процесс *B* вместо того, чтобы отдавать ему ресурс *S*, если бы она знала о предстоящей взаимоблокировке. Работая только с процессами *A* и *C*, мы могли бы получить порядок запросов ресурсов и их возвратов, продемонстрированный на рис. 3.2, *л*, вместо показанного на рис. 3.2, *г*. Такая последовательность действий отражена графами на рис. 3.2, *м—с*, и она не приводит к взаимоблокировке.

После шага *с* процесс *B* может получить ресурс *S*, потому что процесс *A* уже закончил свою работу, а процесс *C* имеет в своем распоряжении все необходимые ему ресурсы. Даже если затем процесс *B*, когда он запросит ресурс *T*, будет заблокирован, система не попадет в тупик. Процесс *B* всего лишь будет ждать завершения работы процесса *C*.

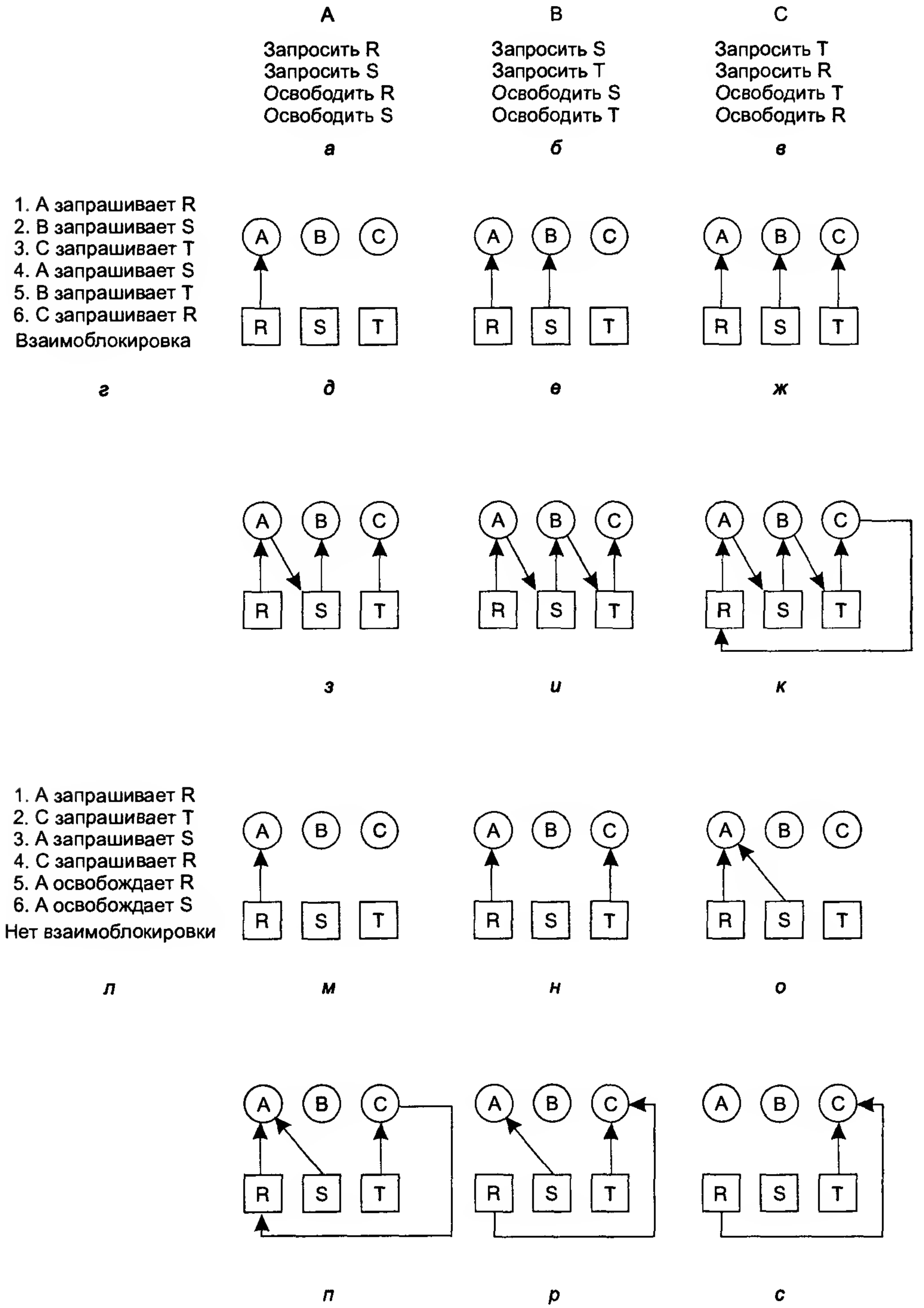


Рис. 3.2. Пример возникновения взаимоблокировки и способы избежать ее

Позже в этой главе мы изучим подробный алгоритм для принятия решений о распределении ресурсов, которые не приведут к взаимоблокировке. В данный момент важно понять, что графы ресурсов являются инструментом, позволяющим нам увидеть, станет ли заданная последовательность запросов/возвратов ресурсов причиной взаимоблокировки. Мы всего лишь шаг за шагом осуществляем запросы и возвраты ресурсов и после каждого шага проверяем граф на содержание циклов. Если они есть, мы зашли в тупик; если нет, значит, взаимоблокировки тоже нет. Хотя мы рассматривали графы ресурсов для случая, когда в системе присутствует по одному ресурсу каждого типа, графы также можно построить для обработки ситуации с несколькими одинаковыми ресурсами [156]. Вообще говоря, при столкновении с взаимоблокировками используются четыре стратегии.

1. Пренебрежение проблемой в целом. Если вы проигнорируете проблему, возможно, затем она проигнорирует вас.
2. Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.
3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.
4. Предотвращение с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Мы по очереди изучим каждый из этих методов в следующих четырех разделах.

Страусовый алгоритм

Самым простым подходом является «страусовый алгоритм»: воткните голову в песок и притворитесь, что проблема вообще не существует. Различные люди отзываются об этой стратегии по-разному. Математики считают ее полностью неприемлемой и говорят, что взаимоблокировки нужно предотвращать любой ценой. Инженеры спрашивают, как часто встает подобная проблема, как часто система попадает в аварийные ситуации по другим причинам и насколько серьезны последствия взаимоблокировок. Если взаимоблокировки случаются в среднем один раз в пять лет, а сбои операционной системы, ошибки компилятора и поломки компьютера из-за неисправности аппаратуры происходят раз в неделю, то большинство инженеров не захотят добровольно уступать в производительности и удобстве для того, чтобы ликвидировать возможность взаимоблокировок.

Для усиления контраста между этими подходами добавим, что большинство операционных систем потенциально страдают от взаимоблокировок, которые даже не обнаруживаются, не говоря уже об автоматическом выходе из тупика. Суммарное количество процессов в системе определяется количеством записей в таблице процесса. Таким образом, ячейки таблицы процесса являются ограниченным ресурсом. Если системный вызов fork получает отказ, потому что таблица целиком заполнена, разумно будет, что программа, вызывающая fork, подождет какое-то время и повторит попытку.

Теперь предположим, что система UNIX имеет 100 ячеек процессов. Работают десять программ, каждой необходимо создать 12 (под)процессов. После образова-

ния каждым процессом девяти процессов 10 исходных и 90 новых процессов заполнят таблицу целиком. Теперь каждый из десяти исходных процессов попадает в бесконечный цикл, состоящий из попыток разветвления и отказов, то есть возникает взаимоблокировка. Вероятность того, что произойдет подобное, минимальна, но это *могло бы* случиться. Должны ли мы отказаться от процессов и вызова fork, чтобы устранить данную проблему?

Максимальное количество открытых файлов также ограниченно размером таблицы i-узлов, следовательно, когда таблица заполняется целиком, возникает та же самая проблема. Пространство для подкачки файлов на диск является еще одним ограниченным ресурсом. Фактически почти каждая таблица в операционной системе представляет собой ресурс, имеющий пределы. Должны ли мы упразднить их все из-за того, что может произойти ситуация, когда в группе из n процессов каждый может потребовать $1/n$ от целого, а затем попытаться получить еще часть?

Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Они исходят из предположения, что большинство пользователей скорее предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем с правилом, по которому всем пользователям разрешается только один процесс, один открытый файл и т. д. Если бы можно было легко устранить взаимоблокировки, не возникло бы столько разговоров на эту тему. Сложность заключается в том, что цена достаточно высока, и в основном она, как мы вскоре увидим, исчисляется в наложении неудобных ограничений на процессы. Таким образом, мы столкнулись с неприятным выбором между удобством и корректностью и множеством дискуссий о том, что более важно и для кого. При всех этих условиях трудно найти верное решение.

Обнаружение и устранение взаимоблокировок

Вторая техника представляет собой обнаружение и восстановление. При использовании этого метода система не пытается предотвратить попадание в тупиковые ситуации. Вместо этого она позволяет взаимоблокировке произойти, старается определить, когда это случилось, и затем совершает некие действия к возврату системы к состоянию, имевшему место до того, как система попала в тупик. В этом разделе мы рассмотрим некоторые из способов обнаружения тупиковых ситуаций и выхода из них.

Обнаружение взаимоблокировки при наличии одного ресурса каждого типа

Начнем с самого простого варианта: в системе существует только один ресурс каждого типа. Подобная система могла бы иметь один сканер, одно устройство для записи компакт-дисков, один плоттер и один накопитель на магнитной ленте, то есть не более чем по одному представителю каждого класса. Другими словами, мы исключаем из рассмотрения системы с двумя одновременно подключенными принтерами. Мы обратимся к ним позже и будем использовать другой метод.

Для такой системы можно сконструировать граф ресурсов вида, продемонстрированного на рис. 3.3. Если этот граф содержит один или больше циклов, значит, произошла взаимоблокировка и блокирован любой процесс, являющийся частью цикла. Если в графе нет циклов, система не попала в тупик.

В качестве примера более сложной системы, чем те, которые мы рассматривали до сих пор, обсудим систему с семью процессами, обозначенными буквами от *A* до *G*, и шестью ресурсами, обозначенными буквами от *R* до *W*. Состояние системы, то есть то, какой процесс владеет каким ресурсом и какой ресурс запрашивается процессом в данный момент, соответствует следующему списку:

1. Процесс *A* занимает ресурс *R* и хочет получить ресурс *S*.
2. Процесс *B* ничего не использует, но хочет получить ресурс *T*.
3. Процесс C ничего не использует, но хочет получить ресурс *S*.
4. Процесс *D* занимает ресурс *U* и хочет получить ресурсы *S* и *T*.
5. Процесс *E* занимает ресурс *T* и хочет получить ресурс *V*.
6. Процесс *F* занимает ресурс *W* и хочет получить ресурс *S*.
7. Процесс *G* занимает ресурс *V* и хочет получить ресурс *U*.

Вопрос: «Заблокирована ли эта система и если да, то какие процессы в этом участвуют?»

Чтобы ответить на этот вопрос, мы можем составить граф ресурсов (рис. 3.3, *а*). Этот граф содержит один цикл, который виден при визуальном обследовании. Цикл показан на рис. 3.3, *б*. Изучая его, можно заметить, что процессы *D*, *E* и *G* заблокированы. Процессы *A*, *C* и *F* не попали в тупик, потому что любому из них можно предоставить ресурс *S*, после чего процесс, получивший ресурс, закончит свою работу и вернет ресурс. Затем два других процесса по очереди могут получить ресурс и также успешно выполнить свою работу.

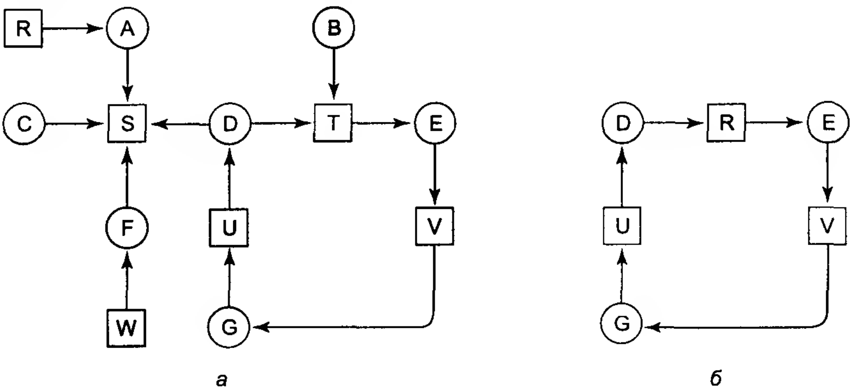


Рис. 3.3. Граф ресурсов (*а*); цикл, извлеченный из *а* (*б*)

Несмотря на то что в простом графе относительно легко зрительно различить взаимоблокировку процессов, для использования такой схемы в настоящей системе нам необходим формальный алгоритм, выявляющий тупики. Известно множе-

ство алгоритмов, обнаруживающих циклы в направленных графах. Ниже мы рассмотрим простой алгоритм, который изучает граф и завершается или когда находит цикл, или когда показывает, что циклов в этом графе не существует. Он использует одну структуру данных — список узлов *L*. Во время работы алгоритма на ребрах графа будет ставиться метка, говорящая о том, что их уже проверили, это делается во избежание повторной проверки.

Алгоритм работает, осуществляя пять перечисленных ниже шагов.

1. Для каждого узла *N* в графе выполняются следующие пять шагов, где *N* является начальным узлом.
2. Задаем начальные условия: *L* — пустой список, все ребра не маркированы.
3. Текущий узел добавляем в конец списка *L* и проверяем количество появлений узла в списке. Если узел присутствует в двух местах, граф содержит цикл (записанный в список *L*) и работа алгоритма завершается.
4. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 5, если нет, то переходим к шагу 6.
5. Случайным образом выбираем любое немаркированное исходящее ребро и отмечаем его. Затем по нему переходим к новому текущему узлу и возвращаемся к шагу 3.
6. Теперь мы зашли в тупик. Удаляем последний узел из списка и возвращаемся к предыдущему узлу, то есть тому, который был текущим перед тупиковым узлом. Обозначаем его текущим узлом и возвращаемся к шагу 3. Если это первоначальный узел, граф не содержит циклов и алгоритм завершается.

Этот алгоритм по очереди берет каждый узел в качестве корня того, что, как он надеется, окажется деревом, и выполняет в дереве поиск в глубину. Если в процессе обхода алгоритм возвращается к уже встречавшемуся узлу, то он нашел цикл. Если алгоритм обходит все ребра из какого-нибудь заданного узла, то он возвращается к предыдущему узлу. Если он возвращается к корню и не может идти дальше, то подграф текущего узла не содержит циклов. Если данное свойство сохраняется для всех узлов, значит, полный граф не содержит циклов, а система не заблокирована.

Чтобы увидеть, как работает описанный алгоритм на практике, воспользуемся графом на рис. 3.3, *а*. Порядок обработки узлов произвольный, поэтому будем исследовать их слева направо и сверху вниз. Тогда алгоритм начнет поиск с узла *R*, затем последовательно с узлов *A*, *B*, *C*, *S*, *D*, *T*, *E*, *F* и т. д. Если мы натолкнемся на цикл, алгоритм остановится.

Мы начинаем с узла *R* и инициализируем *L* как пустой список. Затем добавляем узел *R* в список, переходим к единственно возможному узлу *A*, и его также добавляем к списку *L*, получая *L* = [*R*, *A*]. Из узла *A* следуем к узлу *S*, получая *L* = [*R*, *A*, *S*]. Узел *S* не имеет исходящих ребер, следовательно, это тупик, который заставляет нас вернуться к узлу *A*. Так как у узла *A* тоже нет немаркированных исходящих ребер, мы возвращаемся к узлу *R*, завершая, таким образом, его исследование.

Теперь снова начнем поиск, стартуя с узла *A* и предварительно вернув список *L* в исходное состояние. Эта часть алгоритма тоже быстро остановится, поэтому выполним процесс заново, начиная с узла *B*. Из узла *B* алгоритм будет следовать исходящим ребрам до тех пор, пока не достигнет узла *D*; в это время список будет

таким: $L = [B, T, E, V, G, U, D]$. Теперь мы должны сделать (случайный) выбор. Если выбрать узел S, мы попадаем в тупик и возвращаемся к узлу D. Во второй раз выбираем узел T и получаем измененный список $L = [B, T, E, V, G, U, D, T]$; в этот момент мы обнаруживаем цикл и останавливаем алгоритм.

Этот алгоритм далек от оптимального. Лучшая схема описана в [112]. Тем не менее он демонстрирует существование алгоритма для обнаружения взаимоблокировок.

Обнаружение взаимоблокировок при наличии нескольких ресурсов каждого типа

Когда в системе существует несколько экземпляров некоторых из ресурсов, для обнаружения взаимоблокировок необходим другой метод. Сейчас мы расскажем об основанном на матрицах алгоритме, обнаруживающем тупики среди n процессов, от P_1 до P_n. Пусть m — это число классов ресурсов, причем в системе E_1 ресурсов класса 1, E_2 ресурсов класса 2 и, в общем, E_i ресурсов класса i (где $1 \le i \le m$). E — это **вектор существующих ресурсов**. Он передает общее количество имеющихся в наличии экземпляров каждого ресурса. Например, если класс 1 представляет собой накопители на магнитных лентах, то $E_1 = 2$ означает, что в системе есть два магнитофона.

В любой момент времени некоторые из ресурсов могут оказаться занятыми и, соответственно, недоступны. Пусть A будет **вектором доступных ресурсов**, где A_i равно количеству экземпляров ресурса i, доступных в текущий момент (то есть не использующихся). Если оба накопителя на магнитной ленте заняты, A_1 будет равно 0.

Теперь нам нужны два массива: C — **матрица текущего распределения** и R — **матрица запросов**. i-я строка в матрице C говорит о том, сколько представителей каждого класса ресурсов в данный момент использует процесс P_i. Таким образом, C_{ij} — это количество экземпляров ресурса j, которое занимает процесс i. Аналогично, R_{ij} — это количество экземпляров ресурса j, которые хочет получить процесс P_i. Эти четыре структуры показаны на рис. 3.4.

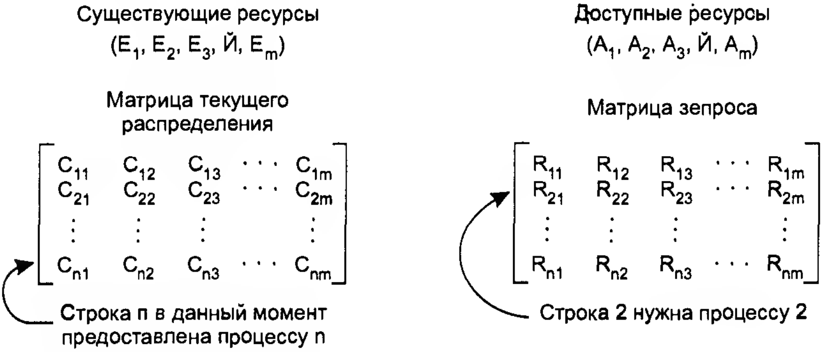


Рис. 3.4. Четыре структуры данных, необходимые для алгоритма обнаружения тупиков

Для этих четырех структур данных существует важный инвариант. В принципе каждый ресурс или занят, или свободен. Это наблюдение означает, что

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j.$$

Другими словами, если сложить все экземпляры ресурса j, предоставленные процессам и доступные в данный момент, то в результате мы получим существующее в системе количество экземпляров этого класса ресурсов.

Алгоритм обнаружения взаимоблокировок основан на сравнении векторов. Определим, что для двух векторов A и B отношение $A \le B$ означает, что каждый элемент вектора A меньше или равен соответствующему элементу вектора B. Математически это запишется так: $A \le B$ тогда и только тогда, когда $A_i \le B_i$ для $1 \le i \le m$.

Пусть в исходном положении все процессы не маркированы. По мере продвижения алгоритма на процессы будет ставиться отметка, служащая признаком того, что они могут закончить свою работу и, следовательно, не находятся в тупике. После завершения алгоритма известно, что любой немаркированный процесс находится в тупиковой ситуации.

Алгоритм обнаружения тупиков состоит из следующих шагов:

1. Ищем немаркированный процесс P_i, для которого i-я строка матрицы R меньше вектора A или равна ему.
2. Если такой процесс найден, прибавляем i-ю строку матрицы C к вектору A, маркируем процесс и возвращаемся к шагу 1.
3. Если таких процессов не существует, работа алгоритма заканчивается.

Завершение алгоритма означает, что все немаркированные процессы, если такие есть, попали в тупик.

На первом шаге алгоритм ищет процесс, который может доработать до конца. Такой процесс характеризуется тем, что все требуемые для него ресурсы должны находиться среди доступных в данный момент ресурсов. Тогда выбранный процесс проработает до своего завершения и после этого вернет ресурсы, которые он занимал, в общий фонд доступных ресурсов. Затем процесс маркируется как законченный. Если окажется, что все процессы могут работать, тогда ни один из них в данный момент не заблокирован. Если некоторые из них никогда не смогут запуститься, значит, они попали в тупик. Несмотря на то что алгоритм не является детерминированным (поскольку он может просматривать процессы в любом допустимом порядке), результат всегда одинаков.

Для иллюстрации работы алгоритма обнаружения тупиков рассмотрим рис. 3.5. Здесь у нас есть три процесса и четыре класса ресурсов, которые мы произвольно назвали так: накопители на магнитной ленте, плоттеры, сканеры и устройство для чтения компакт-дисков. Процесс 1 использует один сканер. Процесс 2 занял два накопителя на магнитной ленте и устройство для чтения компакт-дисков. Процесс 3 занимает плоттер и два сканера. Каждый процесс нуждается в дополнительном устройстве, как показывает матрица R.

Работая с алгоритмом обнаружения взаимоблокировок, мы ищем процесс, чей запрос ресурсов может быть удовлетворен в данной системе. Требования первого процесса нельзя выполнить, потому что в системе нет доступного устройства для чтения компакт-дисков. Второй запрос также нельзя удовлетворить, так как нет

свободных сканеров. К счастью, третий процесс может получить все желаемое, следовательно, он работает, завершается и возвращает все свои ресурсы, давая:

$$A = (2\ 2\ 2\ 0).$$

С этого момента может выполняться процесс 2, по окончании возвращая свои ресурсы в систему. Мы получим:

$$A = (4\ 2\ 2\ 1).$$

Теперь может работать оставшийся процесс. В этой системе не возникает взаимоблокировки.

(Накопители на магнитной ленте, Плоттеры, Сканеры, Компакт-диски)

$$E = (4\quad 2\quad 3\quad 1) \qquad A = (2\quad 1\quad 0\quad 0)$$

Матрица текущего распределения

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Матрица запросов

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Рис. 3.5. Пример использования алгоритма обнаружения тупиков

Теперь обсудим немного измененный вариант ситуации на рис. 3.5. Предположим, что процесс 3, кроме двух накопителей на магнитной ленте и плоттера, нуждается также и в устройстве для чтения компакт-дисков. Тогда ни один из запросов не может быть удовлетворен, значит, вся система находится в тупике.

Мы уже знаем, как обнаружить взаимоблокировки, и появляется вопрос: когда нужно искать их возникновение. Можно проверять систему каждый раз, когда запрашивается очередной ресурс. Это, конечно, позволит обнаружить тупик максимально рано (насколько это возможно), но такая операция может оказаться дорогой в смысле времени загрузки процессора. Альтернативный подход: проверять систему каждые k минут, или, может быть, только когда степень занятости процессора меньше некоторого граничного значения. Учет загрузки процессора имеет смысл, потому что при достаточно большом количестве заблокированных процессов работоспособных процессов в системе останется немного, и процессор часто будет незанятым.

Выход из взаимоблокировки

Предположим, что наш алгоритм обнаружения взаимоблокировок закончился успешно и нашел тупик. Что дальше? Необходимы методы для восстановления и получения в итоге снова работающей системы. В этом разделе мы обсудим раз-

личные способы ликвидации взаимоблокировок. Однако ни один из них не является особо заманчивым.

Восстановление при помощи принудительной выгрузки ресурса

Иногда можно временно отобрать ресурс у его текущего владельца и отдать его другому процессу. Во многих случаях требуется ручное вмешательство, особенно в операционных системах пакетной обработки, работающих на мэйнфреймах.

Например, чтобы забрать лазерный принтер у использующего его процесса, оператор может взять все уже напечатанные листы и сложить их в стопку, соблюдая последовательность их появления из принтера. Затем процесс можно приостановить (пометить как неработоспособный). В этот момент принтер можно предоставить другому процессу. Когда он закончит работу, можно сложить стопку напечатанных листов обратно на выходной поднос принтера и возобновить первоначальный процесс.

Способность забирать ресурс у процесса, отдавать его другому процессу и затем возвращать назад так, что исходный процесс этого не замечает, в значительной мере зависит от свойств ресурса. Выйти из тупика таким образом зачастую трудно или невозможно. Выбор приостанавливаемого процесса главным образом зависит от того, какой процесс владеет ресурсами, которые легко могут быть у него отняты.

Восстановление через откат

Если разработчики системы и машинные операторы знают о том, что есть вероятность появления взаимоблокировок, они могут организовать работу таким образом, чтобы процессы периодически создавали **контрольные точки**. Создание процессом контрольной точки означает, что состояние процесса записывается в файл, в результате чего впоследствии процесс может быть возобновлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, то есть информацию о том, какие ресурсы в данный момент предоставлены процессу. Для большей эффективности новая контрольная точка должна записываться не поверх старой, а в новый файл, так что во время выполнения процесса образуется целая последовательность контрольных точек.

Когда взаимоблокировка обнаружена, достаточно просто понять, какие ресурсы нужны процессам. Чтобы выйти из тупика, процесс, занимающий необходимый ресурс, откатывается к тому моменту времени, перед которым он получил данный ресурс, для чего запускается одна из его контрольных точек. Вся работа, выполненная после этой контрольной копии, теряется (например, выходные данные, напечатанные позднее контрольной копии, отбрасываются и позже печатаются заново). В результате процесс вновь запускается с более раннего момента, когда он не занимал тот ресурс, который теперь предоставляется одному из процессов, попавших в тупик. Если возобновленный процесс снова пытается получить данный ресурс, ему придется ждать того момента, когда ресурс опять станет доступен.

Восстановление путем уничтожения процессов

Грубейший, но одновременно и простейший способ выхода из ситуации взаимоблокировки заключается в уничтожении одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. При небольшом

везении другие процессы смогут продолжить работу. Если первое удаление не помогает, процедуру можно повторять до тех пор, пока цикл наконец не будет разорван.

Можно, наоборот, в качестве жертвы выбрать процесс, не находящийся в цикле, чтобы он освободил свои ресурсы. При этом подходе уничтожаемый процесс выбирается с особой тщательностью, потому что он должен занимать ресурсы, которые нужны некоторым процессам в цикле. Например, один процесс может использовать принтер и требовать плоттер, другой, наоборот, получил плоттер и запрашивает принтер. Оба попали в тупик. Третий процесс может удерживать другие принтер и плоттер и успешно работать. Уничтожение третьего процесса приведет к освобождению этих ресурсов и разрушит взаимоблокировку первых двух процессов.

Там, где это возможно, лучше всего уничтожать те процессы, которые можно запустить с самого начала без всяких болезненных эффектов. Например, процедуру компиляции всегда можно повторить заново, поскольку она всего лишь читает исходный файл и создает объектный файл. Если процедуру компиляции уничтожить в процессе работы, первый ее запуск не повлияет на второй

С другой стороны, процесс, который обновляет базу данных, не всегда можно успешно выполнить во второй раз. Если процесс прибавляет 1 к какой-нибудь записи в базе данных, то его запуск, потом уничтожение, затем повторный запуск приведут к прибавлению к записи 2, что неверно.

Избежание взаимоблокировок

Рассматривая обнаружение взаимоблокировок, мы неявно предполагали, что когда процесс запрашивает ресурсы, он требует их все сразу (матрица R на рис. 3.4). Однако в большинстве систем ресурсы запрашиваются поочередно, по одному. Система должна уметь решать, является ли предоставление ресурса безопасным или нет, и предоставлять его процессу только в первом случае. Таким образом, возникает новый вопрос: существует ли алгоритм, который всегда может избежать ситуации взаимоблокировки, все время делая правильный выбор? Ответом является условное «да» — мы можем избежать тупиков, но только если заранее будет доступна определенная информация. В этом разделе мы изучим способы уклонения от взаимоблокировок с помощью аккуратного предоставления ресурсов.

Траектории ресурсов

Основные алгоритмы, позволяющие предотвращать взаимоблокировки, базируются на концепции безопасных состояний. Перед тем как начать описывать алгоритм, сделаем небольшое отступление, чтобы взглянуть на идею безопасности с точки зрения простого для понимания графического метода. Несмотря на то что графический подход не переносится напрямую в пригодный к употреблению алгоритм, он дает прекрасное интуитивное понимание существа вопроса.

На рис. 3.6 представлена модель для системы с двумя процессами и двумя ресурсами, например принтером и плоттером. Горизонтальная ось отображает номера команд, выполняемых процессом A. По вертикальной оси показаны номера

команд, выполняемых процессом *B*. В команде I_1 процесс *A* запрашивает принтер, в команде I_2 ему требуется плоттер. Принтер и плоттер освобождаются командами I_3 и I_4 соответственно. Процессу *B* необходим плоттер с команды I_5 по команду I_7 и принтер с команды I_6 по команду I_8.

Каждая точка на диаграмме представляет совместное состояние двух процессов. Изначально система находится в точке *p*, когда ни один процесс еще не выполнил ни одну инструкцию. Если планировщик запустит процесс *A* первым, мы попадем в точку *q*, в которой процесс *A* выполнил какое-то количество команд, а процесс *B* еще ничего не сделал. В точке *q* траектория становится вертикальной, показывая, что планировщик решил запустить в работу процесс *B*. При наличии одного процессора все отрезки траектории могут быть только вертикальными или горизонтальными, но не наклонными. Кроме того, движение всегда происходит на север или восток (вверх и вправо), и никогда на юг или запад (вниз и влево), так как процессы не могут работать в обратном направлении.

Когда процесс *A* пересекает линию I_1 на отрезке от точки *r* до точки *s*, он запрашивает и получает принтер. Когда процесс *B* достигает точки *t*, он запрашивает плоттер.

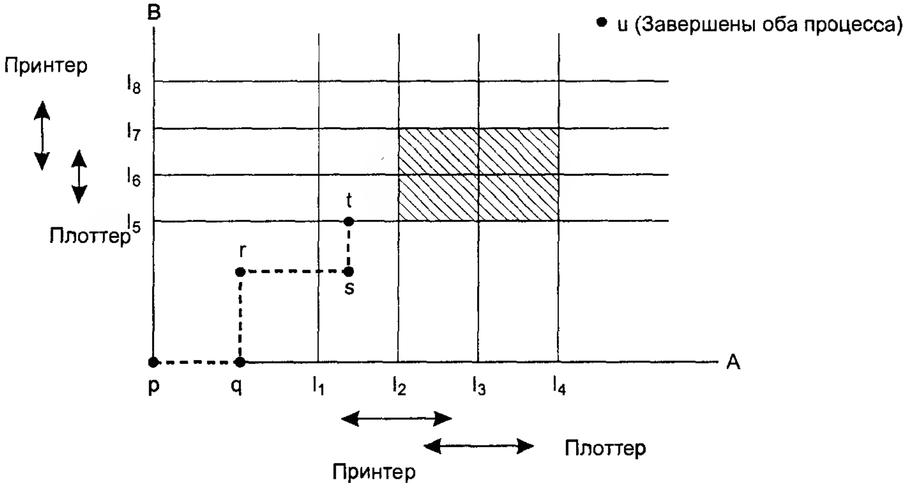


Рис. 3.6. Две траектории ресурсов процессов

Особенно интересны заштрихованные области. Область со штриховкой из верхнего левого угла в правый нижний представляет промежуток времени, когда оба процесса занимают принтер. Правило взаимного исключения делает попадание в эту область невозможным. Вторая заштрихованная область соответствует тому, что оба процесса используют плоттер, и это также невозможно.

Если система войдет в прямоугольник, ограниченный линиями I_1 и I_2 по сторонам и линиями I_5 и I_6 сверху и снизу, она в конце концов доберется до пересечения линий I_2 и I_6, попадет в тупик. В этот момент процесс *A* запросит плоттер, а процесс *B* потребует принтер, но оба ресурса будут к тому времени заняты. Получается, что небезопасным является целый прямоугольник, и в него нельзя входить. В точке *t*

единственно безопасный вариант состоит в том, чтобы оставить процесс *A* работать до тех пор, пока он не достигнет команды I_4. После нее любая траектория дойдет до точки *u*.

Важный для понимания момент заключается в том, что в точке *t* процесс *B* запрашивает ресурс. Система должна принять решение: предоставлять его или нет. Если выдается разрешение, система попадает в небезопасную область и в итоге блокируется. Чтобы избежать тупика, нужно приостановить процесс *B* до тех пор, пока процесс *A* не запросит и не освободит плоттер.

Безопасные и небезопасные состояния

Алгоритмы предотвращения взаимоблокировок, которые мы будем изучать дальше, используют информацию рис. 3.4. В любой момент времени существует текущее состояние, составленное из величин *E*, *A*, *C* и *R*. Говорят, что состояние **безопасно**, если оно не находится в тупике и существует некоторый порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы вдруг захотят немедленно получить свое максимальное количество ресурсов. Проще всего проиллюстрировать эту идею на примере с одним ресурсом. На рис. 3.7, *а* у нас есть состояние, в котором процесс *A* занимает 3 экземпляра ресурса, но ему в итоге могут потребоваться 9 экземпляров. Процесс *B* в настоящий момент занял 2 экземпляра, но позже ему могут понадобиться всего 4. Процесс *C* владеет двумя, но может потребовать еще 5 штук. В системе есть всего 10 экземпляров данного ресурса, 7 из них уже распределены, три пока свободны.

	Имеет	Max
A	3	9
B	2	4
C	2	7

Свободно: 3

а

	Имеет	Max
A	3	9
B	4	4
C	2	7

Свободно: 1

б

	Имеет	Max
A	3	9
B	0	-
C	2	7

Свободно: 5

в

	Имеет	Max
A	3	9
B	0	-
C	7	7

Свободно: 0

г

	Имеет	Max
A	3	9
B	0	-
C	0	-

Свободно: 7

д

Рис. 3.7. Демонстрация того, что состояние *а* безопасно

Состояние на рис. 3.7, *а* безопасно, потому что существует такая последовательность предоставления ресурсов, которая позволяет завершиться всем процессам. А именно, планировщик может просто запустить в работу только процесс *B* на то время, пока он запросит и получит два дополнительных экземпляра ресурса, что приведет к состоянию, изображенному на рис. 3.7, *б*. Когда процесс *B* закончится, мы получим состояние рис. 3.7, *в*. Затем планировщик может запустить процесс *C*, что со временем приведет нас к ситуации рис. 3.7, *г*. По завершении процесса *C* мы получим рис. 3.7, *д*. Теперь процесс *A* наконец может занять необходимые ему шесть экземпляров ресурса и также успешно завершиться. Таким образом, состояние на рис. 3.7, *а* является безопасным, потому что система может избежать тупика с помощью аккуратного планирования процессов.

Теперь предположим, что исходное состояние системы продемонстрировано на рис. 3.8, *а*, но в данный момент процесс *A* запрашивает и получает еще один ресурс, приводя к рис. 3.8, *б*. Сможем ли мы найти последовательность, которая гарантирует работу системы? Давайте попытаемся. Планировщик может дать проработать процессу *B* до того момента, пока он не запросит все свои ресурсы, как показано на рис. 3.8, *в*.

	Имеет	Max
A	3	9
B	2	4
C	2	7

Свободно: 3

а

	Имеет	Max
A	4	9
B	2	4
C	2	7

Свободно: 2

б

	Имеет	Max
A	4	9
B	4	4
C	2	7

Свободно: 0

в

	Имеет	Max
A	4	9
B	4	4
C	2	7

Свободно: 4

г

Рис. 3.8. Демонстрация того, что состояние *б* небезопасно

В итоге процесс *B* успешно завершается и мы получаем ситуацию рис. 3.8, *г*. В этом месте мы застряли: в системе осталось только четыре свободных экземпляра ресурса, а каждому из активных процессов необходимо пять. И не существует последовательности действий, гарантирующей успешное завершение всех процессов. Следовательно, решение о предоставлении ресурса, которое передвинуло систему из положения рис. 3.8, *а* к рис. 3.8, *б*, привело ее из безопасного в небезопасное состояние. Если из ситуации рис. 3.8, *б* запустить процесс *A* или *C*, мы не выйдем из тупика. Теперь, оглядываясь назад, можно уверенно сказать, что нельзя было выполнять запрос процесса *A*.

Следует отметить, что небезопасное состояние само по себе не является тупиком. Начав с рис. 3.8, *б*, система может проработать некоторое время. Фактически даже может успешно завершиться один процесс. Кроме того, возможна ситуация, что процесс *A* сможет освободить один ресурс до следующего своего запроса, позволяя успешно завершиться процессу *C*, а системе избежать взаимной блокировки. Таким образом, разница между безопасным и небезопасным состоянием заключается в следующем: в безопасном состоянии система может *гарантировать*, что все процессы закончат свою работу, а в небезопасном состоянии такой гарантии дать нельзя.

Алгоритм банкира для одного вида ресурсов

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra, [96]) и носит название **алгоритма банкира**. Он представляет собой расширение алгоритма обнаружения тупиков, о котором было рассказано в разделе «Обнаружение взаимоблокировки при наличии одного ресурса каждого типа» данной главы. Модель алгоритма основана на примере банкира в маленьком городке, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведет ли выполнение каждого запроса к небез-

опасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу. На рис. 3.9, *а* мы видим четырех клиентов: *A*, *B*, *C* и *D*, каждый из которых получил определенное количество единиц кредита (например, 1 единица равна 1 К долларов). Банкир знает, что не всем клиентам понадобится их максимальный кредит немедленно, поэтому он зарезервировал только 10 единиц, а не все 22, которые требуются клиентам. (Чтобы провести аналогию с компьютерной системой, считаем, что клиенты — это процессы, единицами, скажем, являются накопители на магнитной ленте, а банкир — это операционная система.)

	Имеет	Max
A	0	6
B	0	5
C	0	4
D	0	7

Свободно: 10

а

	Имеет	Max
A	1	6
B	1	5
C	2	4
D	4	7

Свободно: 2

б

	Имеет	Max
A	1	6
B	2	5
C	2	4
D	4	7

Свободно: 1

в

Рис. 3.9. Три состояния распределения ресурсов: безопасное (а); безопасное (*б*); небезопасное (*в*)

Клиенты вращаются в соответствующем бизнесе, время от времени прося у банка ссуды (то есть запрашивая ресурсы). В некоторый момент возникает ситуация, показанная на рис. 3.9, *б*. Это состояние безопасно, потому что остались две единицы и банкир может задержать все обращения, кроме запросов клиента или процесса *C*, таким образом, позволяя процессу *C* завершиться и вернуть все четыре отданных ему ресурса. Имея на руках четыре единицы, банкир может отдать их или клиенту *D*, или *B*, обеспечивая их необходимыми единицами и т. д.

Рассмотрим, что могло бы произойти, если бы в ситуации на рис. 3.9, *б* был бы удовлетворен запрос еще одной единицы для клиента *B*. Мы попали бы в состояние рис. 3.9, *в*, не являющееся безопасным. Если бы все клиенты вдруг запросили максимальные ссуды, то банкир не смог бы их обеспечить и мы попали бы в тупик. Небезопасное состояние не *обязано* приводить к взаимоблокировке, так как клиентам не обязательно потребуется весь доступный кредит, но банкир не может рассчитывать на такую ситуацию.

Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию. Если да, то процесс получает ресурс, иначе запрос откладывается на более позднее время. Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для завершения работы какого-либо клиента. Если да, то эти ссуды считаются погашенными, после чего проверяется следующий ближайший к пределу займа клиент и т. д. Если, в конце концов, все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить.

Алгоритм банкира для нескольких видов ресурсов

Алгоритм банкира можно обобщить для управления системой с несколькими видами ресурсов. На рис. 3.10 показано, как он работает.

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Распределенные ресурсы

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Ресурсы, которые еще нужны

E = (6342)
P = (5322)
A = (1020)

Рис. 3.10. Алгоритм банкира в системе с несколькими типами ресурсов

На рис. 3.10 изображены две матрицы. Матрица слева показывает, сколько ресурсов каждого вида занимает в настоящее время каждый из пяти процессов. Матрица справа показывает количество ресурсов, которое нужно добавить каждому процессу для успешного завершения. Эти матрицы на рис. 3.4 назывались C и R. Как и в случае одного вида ресурсов, процессы должны точно определять необходимое суммарное количество ресурсов до начала работы для того, чтобы система могла рассчитать правую матрицу в каждый момент времени.

Три вектора, изображенные справа от матриц, показывают, соответственно, существующие ресурсы (вектор E), занятые ресурсы (вектор P) и доступные ресурсы (вектор A). Из вектора E мы видим, что система имеет шесть накопителей на магнитной ленте, три плоттера, четыре принтера и два устройства для чтения компакт-дисков. Из них заняты в данный момент пять накопителей, три плоттера, два принтера и два устройства для чтения компакт-дисков. Чтобы увидеть этот факт, нужно просуммировать четыре столбца, соответствующие ресурсам, в левой матрице. Вектор доступных ресурсов является разницей между тем, что присутствует в системе, и тем, что используется в настоящее время.

Теперь можно изложить алгоритм для проверки безопасности состояния системы.

1. Ищем в матрице R строку, соответствующую процессу, чьи неудовлетворенные потребности ресурсов меньше или равны вектору A. Если такой строки не существует, то система в конце концов попадет в тупик, так как ни один процесс не может проработать до успешного завершения.
2. Допускаем, что процесс, строку которого выбрали в пункте 1, запрашивает все необходимые ресурсы (гарантируется, что это возможно) и заканчивает работу. Отмечаем этот процесс как завершенный и прибавляем все его ресурсы к вектору A.

3. Повторяем шаги 1 и 2 до тех пор, пока или все процессы будут помечены как завершенные — и состояние в этом случае является безопасным, или произойдет взаимоблокировка — тогда состояние небезопасно.

Если на первом шаге можно выбрать несколько процессов, не имеет значения, какой из них будет взят: общий резерв доступных ресурсов или увеличится или, в худшем случае, останется неизменным.

Теперь вернемся к примеру на рис. 3.10. Текущее состояние является безопасным. Предположим, что процесс *B* в данный момент запрашивает принтер. На этот запрос можно ответить положительно, потому что получающееся в результате состояние все еще будет безопасным (процесс *D* может доработать до конца, затем процесс *A* или *E*, затем остальные).

Теперь представим, что после того, как процесс *B* получил один из двух оставшихся принтеров, процесс *E* потребует последний принтер. Удовлетворение этого запроса сократит вектор доступных ресурсов до (1 0 0 0), что приведет к взаимоблокировке процессов. Ясно, что следует отложить на время запрос процесса *E*.

Дейкстра (Dijkstra) впервые опубликовал алгоритм банкира в 1965 году. С тех пор практически каждая книга по операционным системам описывает его в деталях. Различным аспектам этого алгоритма было посвящено бессчетное количество статей. К сожалению, мало у кого из авторов хватило смелости показать, что хотя алгоритм замечателен в теории, на практике он, по существу, бесполезен, потому что нечасто можно определить заранее, сколько ресурсов потребуется процессам в будущем. Кроме того, количество процессов не фиксировано, оно динамически изменяется по мере входа пользователей в систему и выхода из нее. И, более того, ресурсы, про которые считалось, что они доступны, могут внезапно исчезнуть (например, накопитель на магнитной ленте может сломаться). Таким образом, на практике немногие системы, если это вообще имеет место, используют алгоритм банкира для уклонения от взаимоблокировок.

Предотвращение взаимоблокировок

Как мы видели, уклонение от взаимоблокировок, в сущности, невозможно, потому что оно требует наличия никому не известной информации о будущих процессах. Тогда возникает справедливый вопрос: как же реальные системы избегают попадания в тупики? Для того чтобы ответить на этот вопрос, вернемся назад к четырем условиям, сформулированным в [70] (см. раздел «Условия взаимоблокировки» данной главы), и посмотрим, смогут ли они дать нам ключ к разрешению проблемы. Если мы сможем гарантировать, что хотя бы одно из этих условий никогда не будет выполнено, тогда взаимоблокировки станут конструктивно невозможными [150].

Атака условия взаимного исключения

Сначала попробуем атаковать условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, мы никогда не попадем в тупик. Но в равной степени понятно, что если позволить двум процессам одновременно печатать данные на принтере, воцарится хаос. Используя под-

качку[1] выходных данных для печати, несколько процессов могут одновременно генерировать свои выходные данные. В такой модели только один процесс, который фактически запрашивает физический принтер, является демоном[2] принтера. Так как демон не запрашивает никакие другие ресурсы, для принтера мы можем исключить тупики.

К сожалению, не все устройства поддерживают подкачку данных (таблицу процессов невозможно подкачивать с диска). Кроме того, конкуренция за дисковое пространство для подкачки сама по себе может привести к тупику. Что получится, если два процесса заполнили своими выходными данными каждый по половине дискового пространства, отведенного под подкачку данных, и ни один из них не закончил вычисления? Демон может быть запрограммирован так, что начнет печать, не дожидаясь подкачки всех выходных данных, и принтер тогда простоит впустую в том случае, если вычисляющий процесс решил подождать несколько часов после первого пакета выходных данных. По этой причине обычно демоны программируют так, что они начинают печать только после того, как файл выходных данных целиком станет доступен. В этом случае мы получаем два процесса, каждый из которых обработал часть выходных данных, но не все и не может продолжать вычисления дальше. Ни один из двух процессов никогда не завершится, так что произошла взаимоблокировка на диске.

Тем не менее в этом проглядывает росток часто применяющегося решения. Избегайте выделения ресурса, когда это не является абсолютно необходимым, и пытайтесь обеспечить ситуацию, в которой фактически претендовать на ресурс может минимальное количество процессов.

Атака условия удержания и ожидания

Вторoe из условий, сформулированных Коффманом (Coffman) и другими, кажется, все же подает надежду. Если мы сможем уберечь процессы, занимающие некоторые ресурсы, от ожидания остальных ресурсов, мы устраним ситуацию взаимоблокировки. Один из способов достижения этой цели состоит в требовании, следуя которому любой процесс должен запрашивать все необходимые ресурсы до начала работы. Если все ресурсы доступны, процесс получит все, что ему нужно, и сможет работать до успешного завершения. Если один или несколько ресурсов заняты, процессу ничего не предоставляется, и он непременно попадает в состояние ожидания.

Первая проблема при этом подходе заключается в том, что многие процессы не знают, сколько ресурсов им понадобится, до тех пор, пока не начнут работу. На самом деле, если бы они обладали подобными сведениями, то мог бы использоваться и алгоритм банкира. Другая проблема состоит в том, что при этом методе ресурсы не будут использоваться оптимально. Возьмем, например, процесс, кото-

---

[1] Подкачка или спулинг (spooling) — процесс обработки посылаемых на печать документов, которые сохраняются на диске до момента, когда печатающее устройство сможет их обработать. — *Примеч. перев.*

[2] Демон — скрытая от пользователя служебная программа, работающая в фоновом режиме. — *Примеч. перев.*

рый читает данные с входной ленты, анализирует их в течение часа и затем пишет выходную ленту, а заодно и чертит результаты на плоттере. Если все ресурсы нужно запрашивать заранее, то процесс в течение часа не позволит работать накопителю на магнитной ленте и принтеру.

И все-таки некоторые пакетные системы на мэйнфреймах требуют, чтобы пользователи объявляли список всех ресурсов в первой строке каждого задания. Затем система немедленно запрашивает все ресурсы и сохраняет их до окончания задачи. Этот способ накладывает ограничения на деятельность программиста и занимается расточительством ресурсов, зато предотвращает безвыходные тупиковые ситуации.

Немного отличный метод, позволяющий нарушить условие удержания и ожидания, заключается в наложении следующего требования на процесс, запрашивающий ресурс: процесс сначала должен временно освободить все используемые им в данный момент ресурсы. Затем этот процесс пытается сразу получить все необходимое.

Атака условия отсутствия принудительной выгрузки ресурса

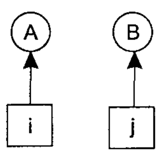
Попытка исключить третье условие (нет принудительной выгрузки ресурса) подает еще меньше надежд, чем устранение второго условия. Если процесс получил принтер и в данный момент печатает выходные данные, насильственное изъятие принтера по причине недоступности требуемого плоттера в лучшем случае сложно, а в худшем — невозможно.

Атака условия циклического ожидания

Остается только одно условие. Циклическое ожидание можно устранить несколькими способами. Один их них: просто следовать правилу, гласящему, что процессу дано право только на один ресурс в конкретный момент времени. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение неприемлемо для процесса, копирующего огромный файл с магнитной ленты на принтер.

Другой способ уклонения от циклического ожидания заключается в поддержке общей нумерации всех ресурсов, как показано на рис. 3.11, *а*. Тогда действует следующее правило: процессы могут запрашивать ресурс, когда хотят этого, но все запросы должны быть сделаны в соответствии с нумерацией ресурсов. Процесс может запросить сначала принтер, затем накопитель на магнитной ленте, но не может сначала потребовать плоттер, а затем принтер.

1. Фотонаборное устройство
2. Сканер
3. Плоттер
4. Накопитель на магнитной ленте
5. Устройство для чтения компакт-дисков



а б

Рис. 3.11. Пронумерованные ресурсы (*а*); граф ресурсов (*б*)

При выполнении такого правила граф распределения ресурсов никогда не будет иметь циклов. Покажем, что это так, в случае двух процессов (рис. 3.11, *б*). Мы можем попасть в тупик, только если процесс *A* запросит ресурс *j*, а процесс *B* обратится к ресурсу *i*. Предположим, что ресурсы *i* и *j* различны, значит, они имеют разные номера. Если $i > j$, тогда процессу *A* не позволяется запрашивать ресурс *j*, потому что его номер меньше, чем номер уже имеющегося у него ресурса. Если же $i < j$, тогда процесс *B* не может запрашивать ресурс *i*, потому что этот номер меньше номера уже занятого им ресурса. Так или иначе, взаимоблокировка невозможна.

При работе с несколькими процессами сохраняется та же самая логика. В каждый момент времени один из предоставленных ресурсов будет иметь наивысший номер. Процесс, использующий этот ресурс, уже никогда не запросит другие занятые ресурсы. Он или закончит свою работу или, в худшем случае, запросит ресурс с еще большим номером, а любой такой ресурс окажется доступен. В итоге процесс завершит работу и освободит свои ресурсы. На этот момент сложится ситуация, когда ресурс с высшим номером уже занят каким-то другим процессом, который также сможет закончить свою работу. То есть существует алгоритм, по которому все процессы завершатся без попадания в тупик.

Вариантом этого алгоритма является схема, в которой отбрасывается требование приобретения ресурсов в строго возрастающем порядке, но сохраняется условие, что процесс не может запросить ресурсы с меньшим номером, чем уже у него имеющиеся. Если процесс на начальной стадии запрашивает ресурсы 9 и 10, затем освобождает их, то это равнозначно тому, как если бы он начал работу заново, поэтому нет причины теперь запрещать ему запрос ресурса 1.

Несмотря на то что систематизация ресурсов с помощью их нумерации устраняет проблему взаимоблокировки, бывают ситуации, когда невозможно найти порядок, удовлетворяющий всех. Когда ресурсы включают в себя области таблицы процессов, дисковое пространство для подкачки данных, закрытые записи базы данных и другие абстрактные ресурсы, число потенциальных ресурсов и вариантов их применений может быть настолько огромным, что никакая систематизация не сможет работать.

В табл. 3.1 подведены итоги различных методов для предотвращения тупиков.

Таблица 3.1. Методы предотвращения тупиков

Условие	Метод
Взаимное исключение	Организовывать подкачку данных
Удержание и ожидание	Запрашивать все ресурсы на начальной стадии
Нет принудительной выгрузки ресурса	Отобрать ресурсы
Циклическое ожидание	Пронумеровать ресурсы и упорядочить

Сопутствующие вопросы

В этом разделе мы обсудим несколько разных вопросов, имеющих отношение к взаимоблокировкам. Сюда входят двухфазовое блокирование, тупик без ресурсов и «голодание».

Двухфазовое блокирование

Хотя и избежание, и предотвращение блокировок в общем случае не являются многообещающими, для отдельных прикладных задач известно немало прекрасных алгоритмов специального назначения. Например, во многих системах баз данных очень часто выполняется операция, которая просит заблокировать несколько записей и затем обновить все заблокированные записи. Когда одновременно работает несколько процессов, появляется реальная опасность взаимоблокировки.

Часто используемый подход называется **двухфазовым блокированием**. В первой фазе, то есть на первом этапе, процесс пытается заблокировать все требуемые записи по одной за раз. Если операция успешна, процесс переходит ко второму этапу, выполняя обновление и освобождение блокировок. Никакой настоящей работы на первом этапе не совершается.

Если во время первой фазы какая-либо необходимая запись оказывается уже заблокированной, процесс просто освобождает все свои блокировки и начинает первую фазу заново. В некотором смысле этот метод похож на схему, в которой запрос всех необходимых ресурсов происходит заранее, или, по крайней мере, перед тем, как произойдет что-то необратимое. В некоторых версиях двухфазового блокирования, если блокировка встретилась во время первой фазы, не происходит возврата ресурсов и возобновления процесса. В таких версиях может возникнуть тупиковая ситуация.

Но эту стратегию нельзя обобщить. В системах реального времени и системах контроля процессов, например, недопустимо частично завершить процесс из-за того, что ресурс недоступен, а потом опять начинать все заново. Также недопустимо перезапускать процесс, если он прочел или написал сообщение в сети, обновил файлы и сделал что-нибудь еще, что не может быть безопасно повторено. Алгоритм работает только в тех ситуациях, когда программист очень тщательно подготовил все таким образом, что программу можно остановить в любой точке первой фазы и запустить заново. Многие программы не могут быть структурированы таким образом.

Тупики без ресурсов

До сих пор все наши рассуждения были сконцентрированы на взаимоблокировках ресурсов. Один процесс хочет получить что-то, что есть у другого процесса, и должен ждать, пока тот не отдаст это что-то. Тем не менее взаимоблокировки также могут происходить и в других ситуациях, включая те, в которых ресурсы вообще не участвуют.

Например, может случиться, что два процесса заблокировали друг друга: каждый ждет, когда другой выполнит некое действие. Такое часто случается с семафорами. В главе 2 мы видели примеры, в которых процесс должен был выполнить системный вызов `down` на двух семафорах, обычно на семафоре *mutex* и еще на одном. Если эту операцию выполнить в неправильном порядке, то в результате получится взаимоблокировка.

Голодание

«Голодание» является проблемой, тесно связанной с взаимоблокировкой. В динамических системах постоянно происходят запросы к ресурсам. Необходима некоторая политика принятия решений о том, когда, кто и какой ресурс получит. Эта политика хотя и кажется разумной, может привести к тому, что некоторые процессы никогда не получат требуемое, хотя они и не будут заблокированы.

Например, рассмотрим процесс предоставления принтера. Представим, что система использует некоторый вид алгоритма, гарантирующий, что предоставление принтера процессу не приводит к взаимоблокировке. Теперь предположим, что несколько процессов одновременно хотят воспользоваться принтером. Какой из них должен получить этот ресурс?

Один возможный алгоритм предоставления ресурсов отдает принтер процессу с наименьшим файлом для печати (предполагается, что подобная информация доступна). Такой подход максимизирует количество счастливых обслуженных клиентов и кажется прекрасным. Теперь рассмотрим, что произойдет в сильно загруженной системе, когда один из процессов должен распечатать огромный файл. Каждый раз, когда принтер освобождается, система выбирает процесс с наиболее коротким файлом. Если в системе есть постоянный поток процессов с небольшими файлами, принтер никогда не будет предоставлен процессу с огромным файлом. Процесс просто «умрет от голода» (будет отложен на неопределенный срок, несмотря на то, что даже не будет заблокирован).

«Голодания» можно избежать, если использовать стратегию распределения ресурсов по принципу «первым пришел — первым обслужен». При таком подходе процесс, ожидающий дольше всех, обслуживается следующим. В итоге любой процесс в некоторый момент станет самым старшим в очереди и, таким образом, получит необходимый ресурс.

Исследования в области взаимоблокировок

Если когда-либо и существовал предмет, на исследования которого не жалели усилий на заре эпохи создания операционных систем, то им были тупиковые ситуации. Так происходило по следующей причине: обнаружение взаимоблокировок является приятной небольшой проблемой из теории графов, в которую имеющие степень в математике студенты могли вонзить свои зубы и пережевывать эту проблему в течение 3-х или 4-х лет. Были изобретены самые разнообразные алгоритмы, каждый последующий все экзотичнее и непрактичнее предыдущего. В конечном итоге исследования в данной области прекратились. Только изредка появляется новая статья, посвященная данной теме (например, [171]). Когда операционные системы хотят обнаружить или предотвратить взаимоблокировку, что некоторые из них и делают, они используют один из методов, обсуждавшихся этой главе.

Однако существует еще одно небольшое изыскание по обнаружению распределенных взаимоблокировок. Мы не будем рассматривать его здесь, потому что (1) это выходит за рамки данной книги и (2) ничто из этого даже отдаленно не практикуется в реальных системах. Кажется, его главная функция — давать работу исследователям в области теории графов, в противном случае пополнивших бы очереди на бирже труда.

Резюме

Взаимоблокировка — это потенциальная проблема любой операционной системы. Она происходит, когда в группе процессов каждый получает монопольный доступ к некоторому ресурсу и каждому требуется еще один ресурс, принадлежащий другому процессу в группе. Все процессы оказываются заблокированными, и ни один никогда не сможет заработать снова.

Взаимоблокировки можно избежать, отслеживая, которое состояние является безопасным, а которое нет. Безопасное состояние — это то, в котором существует последовательность действий, гарантирующая, что все процессы смогут окончить свою работу. В небезопасном состоянии таких обязательств дать нельзя. Алгоритм банкира избегает тупиков, не выполняя запрос, если тот приводит систему в небезопасное состояние.

Взаимоблокировки можно предотвратить структурно, построив систему таким образом, что тупиковая ситуация никогда не возникнет по построению. Например, если позволить процессу использовать только один ресурс в любой момент времени, не выполнится необходимое для возникновения тупиков условие циклического ожидания. Взаимоблокировки также можно предотвратить, если перенумеровать все ресурсы и затем требовать от процессов создания запросов в строго возрастающем порядке. «Голодания» можно избежать, если использовать стратегию распределения ресурсов «первым пришел — первым обслужен».

Вопросы

1. Приведите пример взаимоблокировки, взятый из области политики.
2. Студенты, работая на персональных компьютерах в лаборатории, посылают свои файлы для печати на сервер, записывающий файлы в область подкачки данных на жесткий диск. При каких условиях может произойти взаимоблокировка, если дисковое пространство для подкачки данных печати ограничено? Как можно избежать взаимоблокировки?
3. Какие ресурсы в предыдущем вопросе являются выгружаемыми, а какие — нет?
4. В листинге 3.1 ресурсы возвращаются в порядке, обратном их получению. Будет ли с тем же успехом работать эта схема, если возвращать их в другом порядке?

5. Рисунок 3.1 демонстрирует концепцию графа ресурсов. Существует ли недопустимый граф, то есть такой граф, структура которого нарушает модель, использованную нами для распределения ресурсов? Если да, приведите пример.
6. Обсуждая страусовый алгоритм, мы упоминали про возможность того, что ячейки в таблице процессов или в других системных таблицах заполнятся целиком. Можете ли вы предложить метод, дающий возможность системному администратору вывести систему из такой ситуации?
7. Рассмотрим рис. 3.2. Предположим, что на шаге *n* процесс *C* вместо ресурса *R* запрашивает ресурс *S*. Приведет ли это к взаимоблокировке? А если он запросит оба ресурса, то есть и *S*, и *R*?
8. На перекрестках со знаком STOP по всем четырем направлениям существует правило: каждый водитель уступает дорогу машине справа. Это правило не применимо, когда к перекрестку одновременно подъезжают четыре автобуса. К счастью, люди иногда способны действовать более разумно, чем компьютеры, и подобная проблема обычно решается, когда один из водителей подает знак ехать машине слева от себя. Можете ли вы провести аналогию между таким образом действий и способами выхода из тупиков, описанными в разделе «Выход из взаимоблокировки»? Почему эту ситуацию так трудно применить к компьютерной системе?
9. Предположим, что на рис. 3.4 $C_{ij} + R_{ij} < E_j$ для некоторых *i*. Какое значение имеет это неравенство для всех процессов, заканчивающихся без блокировок?
10. Все траектории на рис. 3.6 горизонтальны или вертикальны. Можете ли вы представить себе условия, при которых также были бы возможны наклонные траектории?
11. Может ли схема траектории ресурсов на рис. 3.6 также использоваться для иллюстрации проблемы тупиков с тремя процессами и тремя ресурсами? Если да, то как? Если нет, то почему?
12. Теоретически график траектории ресурсов мог бы использоваться для избежания тупиков. При умном планировании операционная система могла бы уклоняться от попадания в небезопасные области. Предложите практическую проблему с фактическим выполнением этого.
13. Внимательно посмотрите на рис. 3.9, *б*. Если процесс *D* запросит еще одну единицу, приведет это к безопасному состоянию или к небезопасному? Что будет, если запрос поступит от процесса *C* вместо процесса *D*?
14. Может ли система находиться в состоянии, не являющимся ни состоянием взаимоблокировки, ни безопасным состоянием? Если да, приведите пример. Если нет, докажите, что все состояния либо являются тупиками, либо они безопасны.
15. Пусть в системе существует два процесса и три одинаковых ресурса. Каждому процессу требуется максимум два ресурса. Возможна ли взаимоблокировка? Объясните ваш ответ.

16. Снова рассмотрим предыдущий вопрос, но пусть теперь в системе будет p процессов, каждому нужно максимум m ресурсов, а всего доступно в системе r ресурсов. Какое условие должно выполняться, чтобы в системе не было тупиков?
17. Предположим, что процесс A на рис. 3.10 запрашивает последний накопитель на магнитной ленте. Приведет ли это к взаимоблокировке?
18. У компьютера есть шесть накопителей на магнитной ленте и n процессов, соревнующихся за право их использовать. Каждому процессу может потребоваться два накопителя. При каких значениях n в системе не будет тупиков?
19. Алгоритм банкира работает в системе, где есть m классов ресурсов и n процессов. При m и n стремящихся к бесконечности количество операций, которое нужно выполнить для проверки безопасности состояния, пропорционально $m^a n^b$. Что представляют собой величины a и b?
20. В системе есть четыре процесса и пять ресурсов, которые можно предоставить процессам. Текущее распределение ресурсов и максимальное их количество, необходимое процессам, следующее:

	Предоставлено	**Максимум**	**Доступно**
Процесс A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Процесс B	2 0 1 1 0	2 2 2 1 0	
Процесс C	1 1 0 1 0	2 1 3 1 0	
Процесс D	1 1 1 1 0	1 1 2 2 1	

Каково наименьшее значение величины x, при котором это состояние является безопасным?

21. Распределенная система, использующая почтовые ящики, имеет два примитива межпроцессного взаимодействия: `send` (послать) и `receive` (получить). Второй примитив указывает процесс, от которого следует получить сообщение, и блокируется, если сообщения от процесса недоступны, даже несмотря на то, что могут ожидаться сообщения от других процессов. Здесь нет общих ресурсов, но процессам необходимо часто связываться друг с другом относительно других вопросов. Возможна ли взаимоблокировка? Аргументируйте ответ.
22. Есть два процесса A и B, каждому нужны три записи в базе данных: 1, 2 и 3. Если процесс A обратится к записям в порядке 1, 2, 3 и процесс B запросит их в том же самом порядке, взаимоблокировка невозможна. Однако если процесс B обратится к записям в порядке 3, 2, 1, то появление взаимоблокировки станет возможным. При наличии трех ресурсов в системе есть 3! (что равно 6) возможных комбинации того, как каждый процесс может запросить ресурсы. Какая часть от количества всех комбинаций гарантирует отсутствие тупиков в системе?
23. Теперь рассмотрим ситуацию предыдущего вопроса, но с использованием двухфазового блокирования. Устранит ли это потенциальную возможность взаимоблокировок? Обладает ли оно какими-либо другими нежелательными характеристиками? Если да, то какими?

24. Сотни одинаковых процессов в электронных системах перевода фондов работают следующим образом. Каждый процесс читает входную строку, определяющую количество денег для перевода, кредитовый и дебетовый счета. Затем он блокирует оба счета и переводит деньги, а после завершения перевода снимает блокировку. При параллельно работающем большом количестве процессов существует опасность, что, имея заблокированным счет x, процесс будет неспособен заблокировать счет y, потому что счет y уже окажется заблокированным процессом, в данный момент ожидающим счет x. Разработайте схему действия, избегающую взаимоблокировок. Не освобождайте запись счета до тех пор, пока вы не закончите транзакцию. (Иначе говоря, не позволяются решения, в которых один счет блокируется и затем немедленно освобождается, если другие счета заблокированы.)
25. Один из способов предотвращения взаимоблокировок заключается в нарушении условия удержания и ожидания. В тексте главы предлагалось, что перед запросом нового ресурса процесс должен сначала освободить все ресурсы, уже занятые им (считается, что это возможно). Однако если действовать таким образом, появляется опасность, что процесс может получить новый ресурс, но при этом потерять некоторые из уже имеющихся, необходимых для завершения процесса. Усовершенствуйте эту схему.
26. Студент, получивший работу в области взаимоблокировок, придумал следующий замечательный метод устранения тупиков. Когда процесс запрашивает ресурс, он указывает временной предел. Если процесс блокируется из-за недоступности ресурса, запускается таймер. Если временной предел превышен, процесс разблокируется, и ему позволяется работать снова. Если бы вы были профессором, какую бы оценку вы поставили за этот план, и почему?
27. Золушка и Принц расторгают брак. Чтобы разделись свое имущество, они согласились на следующий алгоритм. Каждое утро любой из них может послать письмо адвокату другого, в котором запрашивает один предмет имущества. Поскольку день уходит на доставку писем, они пришли к соглашению, что если оба обнаруживают, что запросили один и тот же предмет в один и тот же день, на следующий день они посылают письмо с отменой запроса. Среди прочего имущества у них есть собака Вуфер, конура Вуфера, их канарейка Твитер и клетка Твитера. Животные любят свои жилища, поэтому было принято соглашение, что любой вариант раздела имущества, отделяющий животное от его дома, является недействительным, после которого весь раздел имущества требуется начать заново. И Золушка, и Принц отчаянно хотят заполучить Вуфера. Поскольку они могут уехать (отдельно друг от друга) в отпуск, каждый супруг запрограммировал персональный компьютер для обработки переговоров. Когда они возвращаются из отпусков, компьютеры все еще ведут переговоры. Почему? Возможна ли взаимоблокировка? Возможно ли «голодание»? Аргументируйте ответ.
28. Студент, специализирующийся на антропологии и изучавший в качестве неосновной дисциплины вычислительную технику, начал исследовательский проект, чтобы увидеть, могут ли африканские бабуины обучаться ситуации со взаимоблокировками. Он определяет место глубокого каньона и при-

крепляет канат через него, так что бабуины могут переправиться через пропасть на руках. Несколько бабуинов могут переправляться одновременно при условии, что они все будут двигаться в одном направлении. Если бабуины, перемещающиеся в западном и восточном направлении, одновременно залезут на канат, в результате получится взаимоблокировка (бабуины застрянут на середине), потому что один бабуин не может перелезть через другого, пока они оба висят над каньоном. Если бабуин хочет пересечь каньон, он должен проверить, что в данный момент нет других бабуинов, пересекающих каньон в противоположном направлении. Напишите программу, использующую семафоры, избегающую взаимоблокировок. Не думайте о группах бабуинов, движущихся на восток, целиком остановитесь на бабуинах, движущихся на запад.

29. Решите еще раз предыдущую задачу, но теперь попробуйте избежать «голодания». Когда бабуин, желающий перейти на восток, подходит к канату и видит бабуина, пересекающего каньон на запад, он ждет освобождения каната. Но с этого момента бабуинам с востока не разрешается начинать переход каньона до тех пор, пока, по крайней мере, один бабуин не перейдет каньон в противоположном направлении.

30. Составьте программу, моделирующую алгоритм банкира. Ваша программа должна выполнять цикл по каждому клиенту банка, получая запрос и оценивая, является он безопасным или нет. Выведите протокол запросов и принятых решений в файл.

Глава 4
Управление памятью

Память представляет собой важный ресурс, требующий тщательного управления. Несмотря на то что в наши дни память среднего домашнего компьютера в тысячи раз превышает ресурсы IBM 7094 — машины, бывшей в начале 60-х годов самой мощной в мире, — программы все равно увеличиваются в размере быстрее, чем память. Перефразированный закон Паркинсона гласит: «Программы расширяются, стремясь заполнить весь объем памяти, доступный для их поддержки». В этой главе мы рассмотрим, как операционная система управляет памятью.

В идеале каждый программист хотел бы иметь неограниченную в размере и скорости память, при этом также являющуюся энергонезависимой, то есть сохраняющую свое содержимое при выключении электричества — отключении от источников питания. Раз уж мы взялись за эту тему, то почему бы заодно не помечтать о дешевой памяти? К сожалению, технологии не могут обеспечить подобную память. Вследствие этого память в компьютерах имеет **иерархическую структуру**. Небольшая часть ее представляет собой очень быструю, дорогую, энергозависимую (то есть теряющую информацию при выключении питания) кэш-память. Кроме того, компьютеры обладают десятками мегабайт среднескоростной, имеющей среднюю цену, также энергозависимой оперативной памяти ОЗУ (RAM) и десятками или сотнями гигабайт медленного, дешевого, энергонезависимого пространства на жестком диске. Одной из задач операционной системы является координация использования всех этих составляющих памяти.

Часть операционной системы, отвечающая за управление памятью, называется **модулем управления памятью** или **менеджером памяти**. Он следит за тем, какая часть памяти используется в данный момент, а какая — свободна; при необходимости выделяет память процессам и по их завершении освобождает ресурсы; управляет обменом данных[1] между оперативной памятью и диском, если память слишком мала для того, чтобы вместить все процессы.

В этой главе мы изучим несколько различных схем управления памятью, от самой простой до весьма сложной и запутанной. Но начнем мы с самого начала, и прежде всего рассмотрим наиболее элементарную систему управления памятью, а затем постепенно перейдем ко все более и более совершенным конструкциям.

В первой главе мы обращали внимание на то, что в компьютерном мире история имеет тенденцию к повторениям, и хотя простейшие схемы управления памятью больше не используются в настольных персональных компьютерах, они все

---

[1] Последняя операция называется подкачкой данных с диска или свопингом (swapping). — *Примеч. перев.*

еще работают в карманных компьютерах, встроенных системах и смарт-картах. Именно по этой причине их до сих пор стоит изучать.

Основное управление памятью

Системы управления памятью можно разделить на два класса: перемещающие процессы между оперативной памятью и диском во время их выполнения (то есть осуществляющие подкачку процессов целиком (swapping) или использующие страничную подкачку (paging)) и те, которые этого не делают. Второй вариант проще, поэтому начнем с него, а два упомянутых выше вида подкачки мы изучим позже в этой же главе. Читая главу 4, следует помнить, что обычный и постраничный варианты подкачки в значительной степени являются искусственными процессами, вызванными отсутствием достаточного количества оперативной памяти для одновременного хранения всех программ. Если же когда-нибудь оперативная память настолько увеличится в размерах, что ее будет достаточно, аргументы в пользу той или иной схемы управления памятью могут стать устаревшими.

С другой стороны, выше уже упоминалось, что программное обеспечение растет еще быстрее, чем память; поэтому вполне возможно, что потребность в рациональном и эффективном управлении памятью будет существовать всегда. В 80-е годы многие университеты использовали системы разделения времени для работы десятков (более-менее довольных) пользователей на машинах VAX с объемом памяти 4 Мбайт. Сейчас компания Microsoft рекомендует для индивидуальной работы в системе Windows 2000 устанавливать на компьютер, по меньшей мере, 64 Мбайт оперативной памяти. Дальнейшее развитие в сторону мультимедийных систем накладывает еще большие требования на память. Таким образом, весьма вероятно, что качество управления этой частью компьтера будет актуальным по крайней мере в течение следующего десятилетия.

Однозадачная система без подкачки на диск

Самая простая из возможных схем управления памятью заключается в том, что в каждый конкретный момент времени работает только одна программа, при этом память разделяется между программами и операционной системой. На рис. 4.1 показаны три варианта такой схемы. Операционная система может находиться в нижней части памяти, то есть в ОЗУ (оперативное запоминающее устройство, RAM (Random Access Memory — память с произвольным доступом)) — см. рис. 4.1, *а*. Или же операционная система может располагаться в самой верхней части памяти — в ПЗУ (постоянное запоминающее устройство, ROM (Read-Only Memory — память только для чтения)), как продемонстрировано на рис. 4.1, *б*. И третий способ: драйверы устройств могут находиться наверху в ПЗУ, а остальная часть системы — в ОЗУ, расположенной ниже, как показано на рис. 4.1, *в*. Первая модель раньше применялась на мэйнфреймах и мини-компьютерах, но в настоящее время практически не употребляется. Вторая схема сейчас используется на некоторых карманных компьютерах и встроенных системах, а третья модель устанавливалась на ранних персональных компьютерах (например, работающих с MS-DOS), при

этом часть системы в ПЗУ носила название **BIOS** (Basic Input Output System — базовая система ввода-вывода).

Когда система организована таким образом, в каждый конкретный момент времени может работать только один процесс. Как только пользователь набирает команду, операционная система копирует запрашиваемую программу с диска в память и выполняет ее, а после окончания процесса выводит на экран символ приглашения и ждет новой команды. Получив команду, она загружает новую программу в память, записывая ее поверх предыдущей.

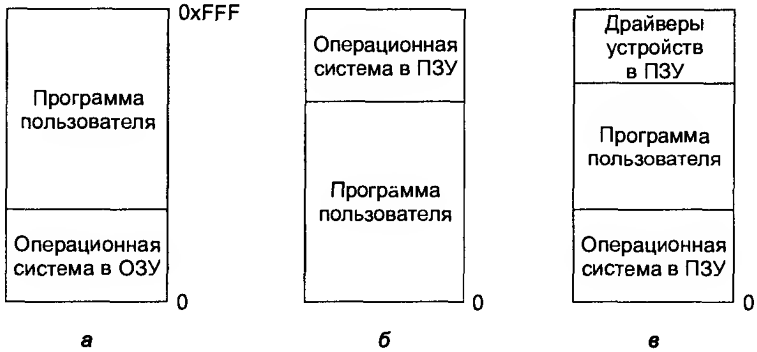


Рис. 4.1. Три простейшие модели организации памяти при наличии операционной системы и одного пользовательского процесса. Существуют также и другие возможные варианты

Многозадачность с фиксированными разделами

Однозадачные системы сложно использовать где-либо еще, кроме простейших встроенных систем. Большинство современных систем позволяет одновременный запуск нескольких процессов. Наличие нескольких процессов, работающих в один момент времени, означает, что когда один процесс приостановлен в ожидании завершения операции ввода-вывода, другой может использовать центральный процессор. Таким образом, многозадачность увеличивает загрузку процессора. Сетевые серверы всегда имеют возможность одновременной работы нескольких процессов (для разных клиентов), но и большинство клиентских машин (то есть настольных компьютеров) в наши дни также имеют эту возможность.

Самый легкий способ достижения многозадачности представляет собой простое разделение памяти на *n* (возможно, не равных) разделов. Такое разбиение можно выполнить, например, вручную при запуске системы.

Когда задание поступает в память, его можно расположить во входной очереди к наименьшему разделу, достаточно большому для того, чтобы вместить это задание. Так как в данной схеме размер разделов неизменен, все пространство в разделе, не используемое работающим процессом, пропадает. На рис. 4.2, *а* показано, как выглядит система с фиксированными разделами и отдельными очередями входных заданий.

Недостаток сортировки входящих работ по отдельным очередям становится очевидным, когда к большому разделу нет очереди, в то время как к маленькому выстроилось довольно много задач; в нашем примере на рис. 4.2, *а* это разделы 1 и 3.

Небольшие задания должны ждать своей очереди, чтобы попасть в память, и это все несмотря на то, что свободна основная часть памяти. Альтернативная схема заключается в организации одной общей очереди для всех разделов, как показано на рис. 4.2, *б*: как только раздел освобождается, задачу, находящуюся ближе всего к началу очереди и подходящую для выполнения в этом разделе, можно загрузить в него и начать ее обработку. Поскольку нежелательно тратить большие разделы на маленькие задачи, существует другая стратегия. Она заключается в том, что каждый раз после освобождения раздела происходит поиск в очереди наибольшего из помещающихся в этом разделе заданий, и именно это задание выбирается для обработки. Заметим, что последний алгоритм дискриминирует мелкие задачи, как недостойные того, чтобы под них отводился целый раздел, хотя обычно крайне желательно предоставить для наименьших задач (часто интерактивных) лучшее, а не худшее обслуживание.

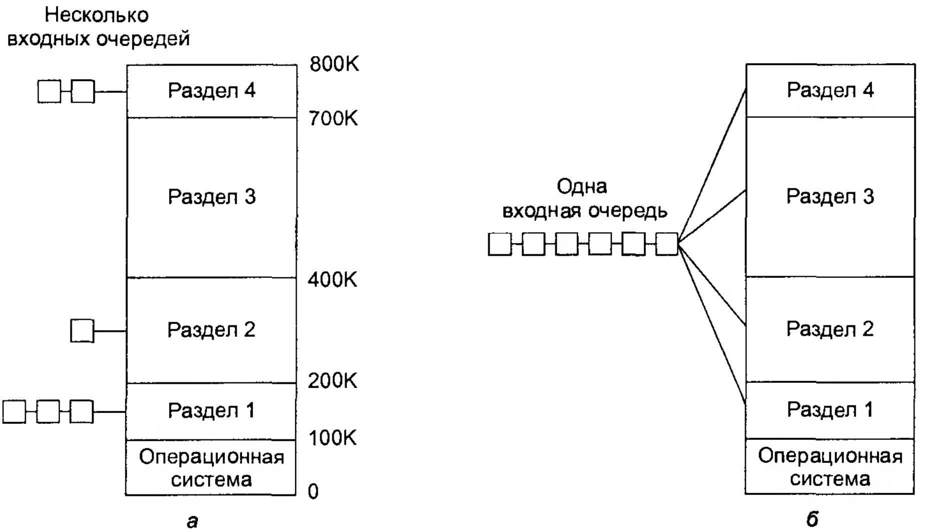


Рис. 4.2. Фиксированные разделы памяти с отдельными входными очередями для каждого раздела (*а*); фиксированные разделы памяти с одной очередью на вход (*б*)

Выйти из положения можно, создав хотя бы один маленький раздел памяти, который позволит выполнять мелкие задания без долгого ожидания освобождения больших разделов.

При другом подходе устанавливается следующее правило: задачу, имеющую право быть выбранной для обработки, можно пропустить не больше k раз. Каждый раз, когда через нее перескакивают, к счетчику добавляется единица. Когда значение счетчика становится равным k, игнорировать задачу более нельзя.

Подобная схема, где утром оператор задает фиксированные разделы и после этого они не изменяются, в течение многих лет использовалась в системах OS/360 на больших мэйнфреймах компании IBM. Она носила название **MFT** (Multiprogramming with a Fixed number of Tasks — мультипрограммирование с фиксированным количеством задач, или OS/MFT). Она легка для понимания и не менее проста в исполнении: входящее задание стоит в очереди до тех пор, пока не

станет доступным соответствующий раздел, затем оно загружается в этот раздел памяти и там работает до завершения процесса. Сейчас очень мало (если они вообще сохранились) операционных систем, поддерживающих такую модель.

Моделирование многозадачности

При использовании многозадачности повышается эффективность загрузки центрального процессора. Грубо говоря, если средний процесс выполняет вычисления только 20 % от того времени, которое он находится в памяти, то при присутствии в памяти одновременно пяти процессов центральный процессор должен быть занят все время. Эта схема слишком оптимистична в отличие от реальной ситуации, поскольку она предполагает, что все пять процессов никогда не ожидают завершения операции ввода-вывода одновременно.

Более совершенная модель рассматривает эксплуатацию центрального процессора с точки зрения теории вероятности. Предположим, что процесс проводит часть p своего времени в ожидании завершения операции ввода-вывода. Если в памяти находится одновременно n процессов, вероятность того, что все n процессов ждут ввод-вывод (в этом случае центральный процессор будет бездействовать), равна p^n. Тогда степень загрузки центрального процессора будет выражаться формулой:

$$\text{Степень загрузки центрального процессора} = 1 - p^n.$$

На рис. 4.3 показана зависимость степени использования центрального процессора от числа n, называемого **степенью многозадачности**.

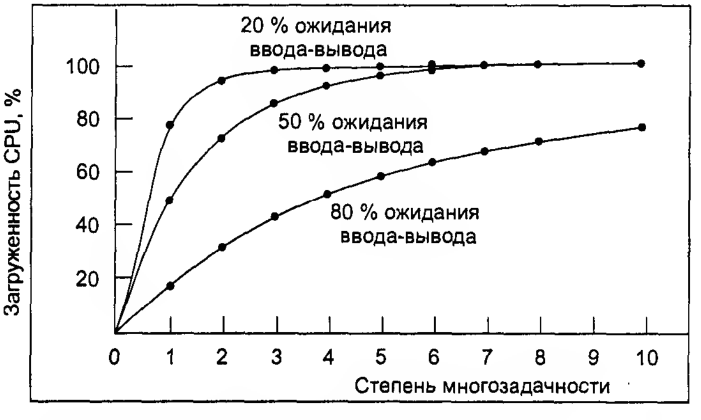


Рис. 4.3. Зависимость степени загрузки центрального процессора от количества процессов в памяти

Из рисунка понятно, что если процессы проводят 80 % своего времени в ожидании завершения операции ввода-вывода, то для того, чтобы получить потерю времени процессора ниже 10 %, в памяти должны одновременно находиться, по меньшей мере, 10 процессов. Когда вы представляете себе, что интерактивный процесс, ожидая, пока пользователь напечатает что-либо на терминале, находится

в состоянии ожидания ввода-вывода, должно быть ясно, что время ожидания ввода-вывода, равное 80 % и больше, не является необычным. Но даже в системах пакетной обработки процессы, выполняющие ввод-вывод в основном с диска, часто имеют такой же или больший процент.

Нужно отметить, что описанная выше вероятностная модель является довольно грубым приближением. Она неявно предполагает, что все *n* процессов независимы, то есть допустима следующая ситуация: в памяти находятся пять процессов, из них три работают, а два ждут. Но когда в системе присутствует один-единственный центральный процессор, он не может одновременно обрабатывать три процесса, поэтому уже готовый к работе процесс обяэан ждать освобождения процессора. Таким образом, в реальности процессы не являются независимыми. Более аккуратную модель можно построить с использованием теории организации очередей, но общая идея, на которую мы обратили внимание — многозадачность позволяет процессам использовать центральный процессор тогда, когда при других обстоятельствах он бы бездействовал, — конечно, останется в силе, даже если кривые на рис. 4.3 немного изменятся.

Хотя модель на рис. 4.3 очень проста, тем не менее она позволяет сделать определенный, хотя и приблизительный, прогноз относительно производительности центрального процессора. Например, предположим, что компьютер имеет 32 Мбайт памяти, 16 Мбайт отдано операционной системе, а каждая программа пользователя занимает по 4 Мбайт. При таких заданных размерах одновременно можно загрузить в память четыре пользовательские программы. При 80 % времени на ожидание ввода-вывода в среднем мы получим загруженность процессора (игнорируя издержки операционной системы) равной $1-0{,}8^4$, или около 60 %. Добавление еще 16 Мбайт памяти позволит системе повысить степень многозадачности от четырех до восьми и таким образом повысить степень загрузки процессора до 83 %. Другими словами, дополнительные 16 Мбайт увеличат производительность на 38 %.

Еще 16 Мбайт могли бы повысить загрузку процессора с 83 до 93 %, таким образом, увеличив производительность всего лишь на 12 %. С помощью этой модели владелец компьютера может решить, что первые 16 Мбайт оперативной памяти — это хорошее вложение капитала, а вторые — нет.

Анализ производительности многозадачных систем

Описанную выше модель также можно применить для анализа систем пакетной обработки. Например, рассмотрим компьютерный центр, в котором среднее время ожидания ввода-вывода задачами равно 80 %. Однажды утром подаются на выполнение 4 задания, как показано на рис. 4.4, *а*. Первая задача, поступившая в 10 утра, требует 4 мин работы процессора. Тогда при 80 % времени ожидания ввода-вывода за каждую минуту своего нахождения в памяти задача использует только 12 с времени процессора, даже если нет никаких других параллельных заданий, также желающих занять процессор. Остальные 48 с процесс проводит в ожидании ввода-вывода. Таким образом, задача должна находиться в памяти по крайней мере

20 мин для того, чтобы процессор сделал работу, требующую на самом деле всего 4 мин, и это все при отсутствии конкуренции на право использования процессора.

Что же происходит дальше? С 10:00 до 10:10 утра в памяти находится целиком первая задача и выполняется половина работы (2 мин работы процессора). Когда в 10:10 поступает второе задание, загрузка процессора увеличивается с 0,20 до 0,36 вследствие более высокой степени многозадачности (см. рис. 4.3). Однако при циклическом планировании каждое задание получает для себя половину времени процессора, поэтому за каждую минуту нахождения в памяти выполняется часть задачи, требующая 0,18 мин работы процессора. Заметим, что добавление второй задачи обходится первой задаче всего в 10 % ее производительности. Время использования процессора за минуту реального времени уменьшилось с 0,20 мин до 0,18 мин.

В 10:15 утра поступает третье задание. В этот момент для первой задачи процессор отработал 2,9 мин, для второй — 0,9 мин. Степень многозадачности теперь равна 3, каждое задание получает для себя 0,16 мин работы процессора за минуту реального времени, как показано на рис. 4.4, *б*. Тогда с 10:15 до 10:20 утра для каждого из трех заданий процессор работает по 0,8 мин. В 10:20 утра поступает четвертая задача. На рис. 4.4, *в* представлена полная последовательность событий.

Задача	Время поступления	Количество минут работы процесса
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

а

	Количество процесса			
	1	2	3	4
Процессор бездействует	.80	.64	.51	.41
Процессор занят	.20	.36	.49	.59
Процессор/процесс	.20	.18	.16	.15

б



в

Рис. 4.4. Время поступления и рабочие требования четырех задач (*а*); загруженность процессора для количества задач от 1 до 4 при 80 % ожидания ввода-вывода (*б*); последовательность событий при поступлении и завершении обработки задач (в). Числа над горизонтальными линиями показывают время процессора в минутах, получаемое каждой задачей в каждом интервале времени

Настройка адресов и защита

Многозадачность вносит две существенные проблемы, требующие решения, — это настройка адресов для перемещения программы в памяти и защита. Посмотрите на рис. 4.2. Из рисунка становится ясно, что разные задачи будут запущены по различным адресам. Когда программа компонуется (то есть в едином адресном пространстве объединяются основной модуль, написанные пользователем процедуры и библиотечные процедуры), компоновщик должен знать, с какого адреса будет начинаться программа в памяти.

Например, предположим, что первая команда представляет собой вызов процедуры с абсолютным адресом 100 внутри двоичного файла, создаваемого компоновщиком. Если эта программа загружается в раздел 1 (по адресу 100 К), команда обратится к абсолютному адресу 100, который находится внутри операционной системы. А нужно вызвать процедуру по адресу 100 К + 100. Если же программа загружается во второй раздел, команду нужно переадресовать на 200 К + 100 и т. д. Эта проблема известна как проблема **перемещения программ в памяти** или **настройки адресов**.

Одним из возможных решений является модификация команд во время загрузки программы в память. В программе, загружаемой в первый раздел, к каждому адресу прибавляется 100 К, в программе, которая загружается во второй раздел, к адресам добавляется 200 К и т. д. Чтобы выполнить подобную настройку адресов во время загрузки, компоновщик должен включить в двоичную программу список или битовый массив с информацией о том, какие слова в программе являются адресами (и их нужно перераспределить), а какие — кодами машинных команд, постоянными или другими частями программы, которые не нужно изменять. Таким образом работает операционная система OS/MFT.

Настройка адресов во время загрузки не решает проблемы защиты. Вредоносные программы всегда могут создать новую команду и перескочить на нее. Поскольку при такой системе программы предпочитают использовать абсолютную адресацию памяти, а не адреса относительно какого-либо регистра, не существует способа, который позволил бы запретить программе построение команды, обращающейся к любому слову в памяти для его чтения или записи. В многопользовательских системах крайне нежелательно разрешать процессам чтение или запись в область памяти, принадлежащую другим пользователям.

Для защиты компьютера 360 компания IBM приняла следующее решение: она разделила память на блоки по 2 Кбайт и назначила каждому блоку 4-битовый защитный код. Регистр PSW (Program Status Word — слово состояния программы) содержал 4-битовый ключ. Аппаратура IBM 360 перехватывала все попытки работающих процессов обратиться к любой части памяти, чей защитный код отличался от содержимого регистра слова состояния программы. Так как только операционная система могла изменять коды защиты и ключи, предотвращалось вмешательство пользовательских процессов в дела друг друга и в работу операционной системы.

Альтернативное решение сразу обеих проблем (защиты и перераспределения) заключается в оснащении машины двумя специальными аппаратными регистрами, называемыми **базовым** и **предельным регистрами**. При планировании процес-

са в базовый регистр загружается адрес начала раздела памяти, а в предельный регистр помещается длина раздела. К каждому автоматически формируемому адресу перед его передачей в память прибавляется содержимое базового регистра. Таким образом, если базовый регистр содержит величину 100 К, команда `CALL 100` будет превращена в команду `CALL 100K+100` без изменения самой команды. Кроме того, адреса проверяются по отношению к предельному регистру для гарантии, что они не используются для адресации памяти вне текущего раздела. Базовый и предельный регистры защищаются аппаратно, чтобы не допустить их изменений пользовательскими программами.

Неудобство этой схемы заключается в том, что требуется выполнять операции сложения и сравнения при каждом обращении к памяти. Операция сравнения может быть выполнена быстро, но сложение — это медленная операция, что обусловлено временем распространения сигнала переноса, за исключением тех случаев, когда употребляется специальная микросхема сложения.

Такая схема использовалась на первом суперкомпьютере в мире CDC 6600. В центральном процессоре Intel 8088 для первых IBM PC применялась упрощенная версия этой модели: были базовые регистры, но отсутствовали предельные. Сейчас такую схему можно встретить лишь в немногих компьютерах.

Подкачка

Организация памяти в виде фиксированных разделов проста и эффективна для работы с пакетными системами. Каждое задание после того, как доходит до начала очереди, загружается в раздел памяти и остается там до своего завершения. До тех пор пока в памяти может храниться достаточное количество задач для обеспечения постоянной занятости центрального процессора, нет причин что-либо усложнять.

Но совершенно другая ситуация сложилась с системами разделения времени или персональными компьютерами, ориентированными на работу с графикой. Оперативной памяти иногда оказывается недостаточно для того, чтобы вместить все текущие активные процессы, и тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить их в память.

Существуют два основных подхода к управлению памятью, зависящие (отчасти) от доступного аппаратного обеспечения. Самая простая стратегия, называемая **свопингом** (swapping) или **обычной подкачкой**, заключается в том, что каждый процесс полностью переносится в память, работает некоторое время и затем целиком возвращается на диск. Другая стратегия, носящая название **виртуальной памяти**, позволяет программам работать даже тогда, когда они только частично находятся в оперативной памяти. Ниже мы изучим свопинг, а вторую стратегию рассмотрим в разделе «Виртуальная память» данной главы.

Работа системы свопинга проиллюстрирована на рис. 4.5. На начальной стадии в памяти находится только процесс *A*. Затем создаются или загружаются с диска процессы *B* и *C*. На рис. 4.5, *г* процесс *A* выгружается на диск. Затем появляется процесс *D*, а процесс *B* завершается. Наконец, процесс *A* снова возвращается

в память. Так как теперь процесс *A* имеет другое размещение в памяти, его адреса должны быть перенастроены или программно во время загрузки в память, или (более заманчивый вариант) аппаратно во время выполнения программы.

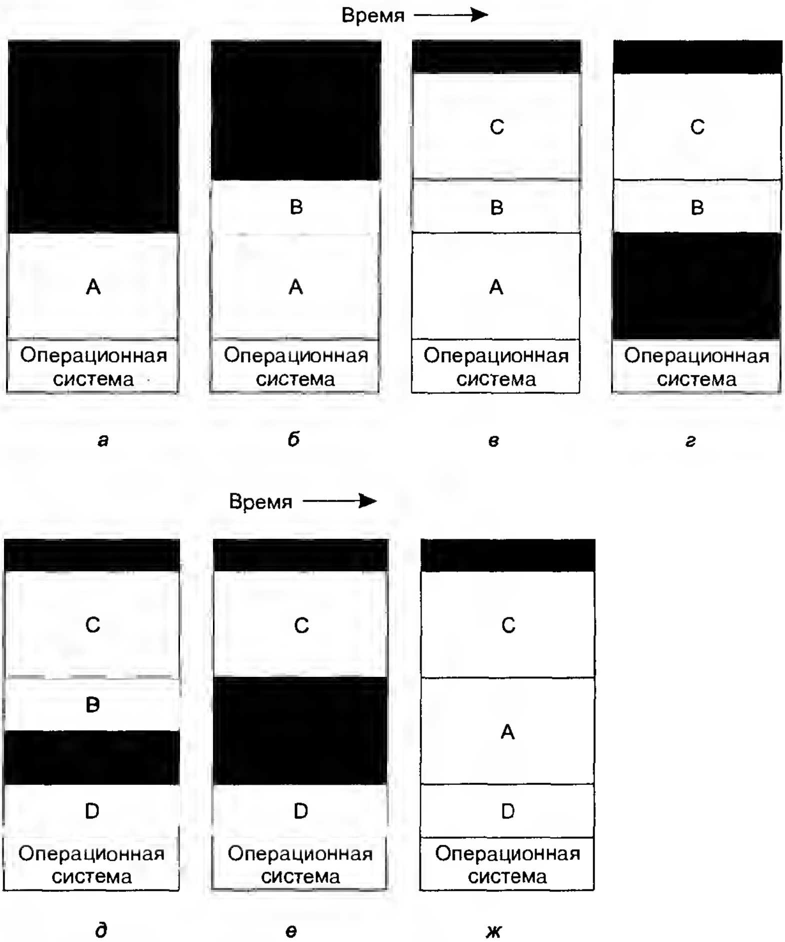


Рис. 4.5. Распределение памяти изменяется по мере того, как процессы поступают в память и покидают ее. Заштрихованы неиспользуемые области памяти

Основная разница между фиксированными разделами на рис. 4.2 и непостоянными разделами на рис. 4.5 заключается в том, что во втором случае количество, размещение и размер разделов изменяются динамически по мере поступления и завершения процессов, тогда как в первом варианте они фиксированы. Гибкость схемы, в которой нет ограничений, связанных с определенным количеством разделов, и каждый из разделов может быть очень большим или совсем маленьким, улучшает использование памяти, но, кроме того, усложняет операции размещения процессов и освобождения памяти, а также отслеживание происходящих изменений.

Когда в результате подкачки процессов с диска в памяти появляется множество неиспользованных фрагментов, их можно объединить в один большой участок, передвинув все процессы в сторону младших адресов настолько, насколько это возможно. Такая операция называется **уплотнением** или **сжатием памяти**. Обычно ее не выполняют, потому что на нее уходит много времени работы процессора. Например, на машине с 256 Мбайт оперативной памяти, которая может копировать 4 байта за 40 нс, уплотнение всей памяти займет около 2,7 с.

Еще один момент, на который стоит обратить внимание: сколько памяти должно быть предоставлено процессу, когда он создается или скачивается с диска? Если процесс имеет фиксированный никогда не изменяющийся размер, размещение происходит просто: операционная система предоставляет точно необходимое количество памяти, ни больше, ни меньше, чем нужно.

Однако если область данных процесса может расти, например, в результате динамического распределения памяти из кучи[1], как происходит во многих языках программирования, проблема предоставления памяти возникает каждый раз, когда процесс пытается увеличиться. Когда участок неиспользованной памяти расположен рядом с процессом, его можно отдать в пользу процесса, таким образом, позволив процессу вырасти на размер этого участка. Если же процесс соседствует с другим процессом, для его увеличения нужно или переместить достаточно большой свободный участок памяти, или перекачать на диск один или больше процессов, чтобы создать незанятый фрагмент достаточного размера. Если процесс не может расти в памяти, а область на диске, предоставленная для подкачки, переполнена, процесс будет вынужден ждать освобождения памяти или же будет уничтожен.

Если предположить, что большинство процессов будут увеличиваться во время работы, вероятно, сразу стоит предоставлять им немного больше памяти, чем требуется, а всякий раз, когда процесс скачивается на диск или перемещается в памяти, обрабатывать служебные данные, связанные с перемещением или подкачкой процессов, больше не умещающихся в предоставленной им памяти. Но когда процесс выгружается на диск, должна скачиваться только действительно используемая часть памяти, так как очень расточительно также перемещать и дополнительную память. На рис. 4.6, *а* можно увидеть конфигурацию памяти с предоставлением пространства для роста двух процессов.

Если процесс может иметь два увеличивающихся сегмента, например сегмент данных, используемый как куча для динамически назначаемых и освобождаемых переменных, и сегмент стека для обычных локальных переменных и возвращаемых адресов, предлагается альтернативная схема распределения памяти, показанная на рис. 4.6, *б*. Здесь мы видим, что у каждого процесса вверху предоставленной ему области памяти находится стек, который расширяется вниз, и сегмент данных, расположенный отдельно от текста программы, который увеличивается вверх. Область памяти между ними разрешено использовать для любого сегмента. Если ее становится недостаточно, то процесс нужно или перенести на другое, большее свободное место, или выгрузить на диск до появления свободного пространства необходимого размера, или уничтожить.

---

[1] Кучей (heap) называется область памяти, выделяемая программе для динамически размещаемых структур данных. — *Примеч. перев.*

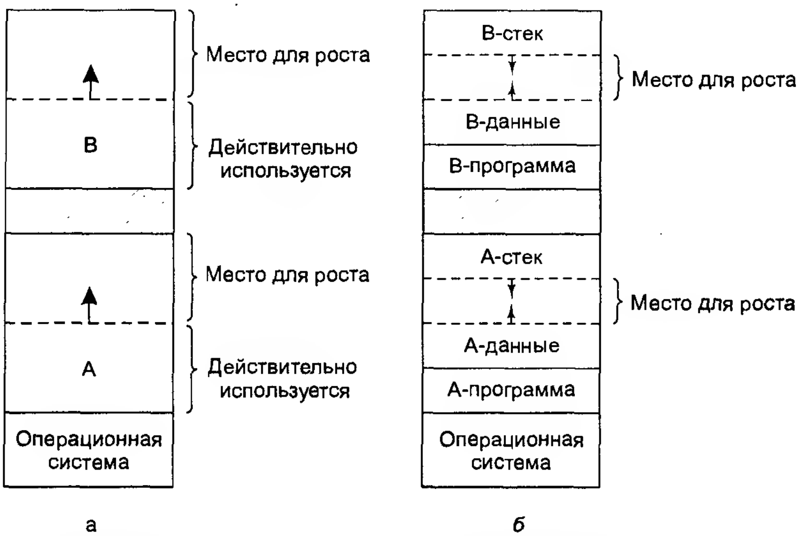


Рис. 4.6. Предоставление пространства для роста области данных (а); предоставление пространства для роста стека и области данных (*б*)

Управление памятью с помощью битовых массивов

Если память выделяется динамически, этим процессом должна управлять операционная система. Существует два способа учета использования памяти: битовые массивы, иногда называемые битовыми картами, и списки свободных участков. В этом и следующем разделах мы по очереди рассмотрим оба метода.

При работе с битовым массивом память разделяется на единичные блоки размещения размером от нескольких слов до нескольких килобайт. В битовой карте каждому свободному блоку соответствует один бит, равный нулю, а каждому занятому блоку — бит, установленный в 1 (или наоборот). На рис. 4.7 показана часть памяти и соответствующий ей битовый массив. Черточками отмечены единичные блоки памяти. Заштрихованные области (0 в битовой карте) свободны.

Размер единичного блока представляет собой важный вопрос стадии разработки системы. Чем меньше единичный блок, тем больше потребуется битовый массив. Однако даже при маленьком единичном блоке, равном четырем байтам, для 32 битов памяти потребуется 1 бит в карте. Тогда память размером в $32n$ будет использовать n битов в карте, таким образом, битовая карта займет всего лишь 1/33 часть памяти. Если выбираются большие единичные блоки, битовая карта становится меньше, но при этом может теряться существенная часть памяти в последнем блоке каждого процесса (если размер процесса не кратен размеру единичного блока).

Битовый массив предоставляет простой способ отслеживания слов в памяти фиксированного объема, потому что размер битовой карты зависит только от размеров памяти и единичного блока. Основная проблема, возникающая при

этой схеме, заключается в том, что при решении переместить k-блочный процесс в память модуль управления памяти должен найти в битовой карте серию из k следующих друг за другом нулевых битов. Поиск серии заданной длины в битовой карте является медленной операцией (так как искомая последовательность битов может пересекать границы слов в битовом массиве). В этом состоит аргумент против битовых карт.

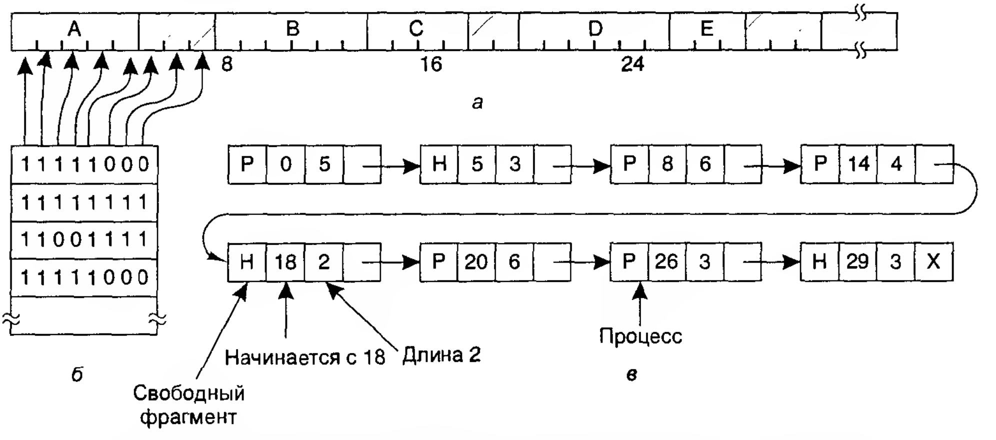


Рис. 4.7. Часть памяти с пятью процессами и тремя свободными областями (*а*); соответствующая битовая карта (*б*); та же информация в виде списка (*в*)

Управление памятью с помощью связных списков

Другой способ отслеживания состояния памяти предоставляет поддержка связных списков занятых и свободных фрагментов памяти, где сегментом является или процесс, или участок между двумя процессами. Память, показанная на рис. 4.7, *а*, представлена в виде связного списка сегментов на рис. 4.7, *в*. Каждая запись в списке указывает, является ли область памяти свободной (H, от hole — дыра) или занятой процессом (P, process); адрес, с которого начинается эта область; ее длину; содержит указатель на следующую запись.

В нашем примере список отсортирован по адресам. Такая сортировка имеет следующее преимущество: когда процесс завершается или скачивается на диск, изменение списка представляет собой несложную операцию. Закончившийся процесс обычно имеет двух соседей (кроме тех случаев, когда он находится на самом верху или на дне памяти). Соседями могут быть процессы или свободные фрагменты, что приводит к четырем комбинациям, показанным на рис. 4.8. На рис. 4.8, *а* корректировка списка требует замены P на H. На рис. 4.8, *б*, *в* две записи соединяются в одну, а список становится на запись короче. На рис. 4.8, *г* объединяются три записи, а из списка удаляются два пункта. Так как ячейка таблицы процессов для завершившегося процесса обычно будет непосредственно указывать на запись в списке для этого процесса, возможно, удобнее иметь список с двумя связями, чем с одной (последний показан на рис. 4.7, *в*). Такая структура упрощает поиск предыдущей записи и оценку возможности соединения.

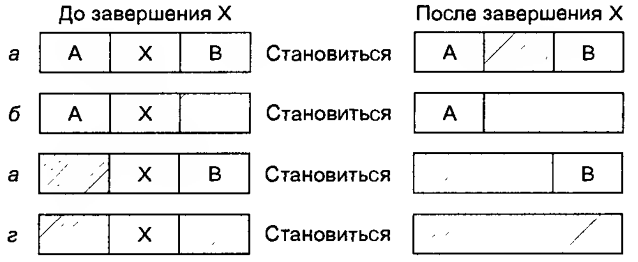


Рис. 4.8. Четыре комбинации соседей для завершения процесса X

Если процессы и свободные участки хранятся в списке, отсортированном по адресам, существует несколько алгоритмов для предоставления памяти процессу, создаваемому заново (или для существующих процессов, скачиваемых с диска). Допустим, менеджер памяти знает, сколько памяти нужно предоставить. Простейший алгоритм представляет собой выбор **первого подходящего участка**. Менеджер памяти просматривает список областей до тех пор, пока не находит достаточно большой свободный участок. Затем этот участок делится на две части: одна отдается процессу, а другая остается неиспользуемой. Так происходит всегда, кроме статистически нереального случая точного соответствия свободного участка и процесса. Это быстрый алгоритм, потому что поиск уменьшен настолько, насколько возможно.

Алгоритм **«следующий подходящий участок»** действует с минимальными отличиями от правила «первый подходящий». Он работает так же, как и первый алгоритм, но всякий раз, когда находит соответствующий свободный фрагмент, он запоминает его адрес. И когда алгоритм в следующий раз вызывается для поиска, он стартует с того самого места, где остановился в прошлый раз вместо того, чтобы каждый раз начинать поиск с начала списка, как это делает алгоритм «первый подходящий». Моделирование работы алгоритма, произведенное Бэйсом, показало, что производительность схемы «следующий подходящий» немного хуже, чем «первый подходящий» [21].

Другой хорошо известный алгоритм называется **«самый подходящий участок»**. Он выполняет поиск по всему списку и выбирает наименьший по размеру подходящий свободный фрагмент. Вместо того чтобы делить большую незанятую область, которая может понадобиться позже, этот алгоритм пытается найти участок, близко подходящий к действительно необходимым размерам.

Чтобы привести пример работы алгоритмов «первый подходящий» и «самый подходящий», снова обратимся к рис. 4.7. Если необходим блок размером 2, правило «первый подходящий» предоставит область по адресу 5, а схема «самый подходящий» разместит процесс в свободном фрагменте по адресу 18.

Алгоритм «самый подходящий» медленнее «первого подходящего», потому что каждый раз он должен производить поиск во всем списке. Но, что немного удивительно, он выдает еще более плохие результаты, чем «первый подходящий» или «следующий подходящий», поскольку стремится заполнить память очень маленькими, бесполезными свободными областями, то есть фрагментирует память. Алгоритм «первый подходящий» в среднем создает большие свободные участки.

Пытаясь решить проблему разделения памяти на практически точно совпадающие с процессом области и маленькие свободные фрагменты, можно задуматься об алгоритме **«самый неподходящий участок»**. Он всегда выбирает самый большой свободный участок, от которого после разделения остается область достаточного размера и ее можно использовать в дальнейшем. Однако моделирование показало, что это также не очень хорошая идея.

Все четыре алгоритма можно ускорить, если поддерживать отдельные списки для процессов и свободных областей. Тогда поиск будет производиться только среди незанятых фрагментов. Неизбежная цена, которую нужно заплатить за увеличение скорости при размещении процесса в памяти, заключается в дополнительной сложности и замедлении при освобождении областей памяти, так как ставший свободным фрагмент необходимо удалить из списка процессов и вставить в список незанятых участков.

Если для процессов и свободных фрагментов поддерживаются отдельные списки, то последний можно отсортировать по размеру, тогда алгоритм «самый подходящий» будет работать быстрее. Когда он выполняет поиск в списке свободных фрагментов от самого маленького к самому большому, то, как только находит подходящую незанятую область, алгоритм уже знает, что она — наименьшая из тех, в которых может поместиться задание, то есть наилучшая. В отличие от схемы с одним списком, дальнейший поиск не требуется. Таким образом, если список свободных фрагментов отсортирован по размеру, схемы «первый подходящий» и «самый подходящий» одинаково быстры, а алгоритм «следующий подходящий» не имеет смысла.

При поддержке отдельных списков для процессов и свободных фрагментов возможна небольшая оптимизация. Вместо создания отдельного набора структур данных для списка свободных участков, как это сделано на рис. 4.7, *в*, можно использовать сами свободные области. Первое слово каждого незанятого фрагмента может содержать размер фрагмента, а второе слово может указывать на следующую запись. Узлы списка на рис. 4.7, *в*, для которых требовались три слова и один бит (Р/Н), больше не нужны.

Еще один алгоритм распределения называется **«быстрый подходящий»**, он поддерживает отдельные списки для некоторых из наиболее часто запрашиваемых размеров. Например, могла бы существовать таблица с *n* записями, в которой первая запись указывает на начало списка свободных фрагментов размером 4 Кбайт, вторая запись является указателем на список незанятых областей размером 8 Кбайт, третья — 12 Кбайт и т. д. Свободный фрагмент размером, скажем, 21 байт, мог бы располагаться или в списке областей 20 Кбайт или в специальном списке участков дополнительных размеров. При использовании правила «быстрый подходящий» поиск фрагмента требуемого размера происходит чрезвычайно быстро. Но этот алгоритм имеет тот же самый недостаток, что и все схемы, которые сортируют свободные области по размеру, а именно: если процесс завершается или выгружается на диск, поиск его соседей с целью узнать, возможно ли их соединение, является дорогой операцией. А если не производить слияния областей, память очень скоро окажется разбитой на огромное число маленьких свободных фрагментов, в которые не поместится ни один процесс.

Виртуальная память

Уже достаточно давно люди впервые столкнулись с проблемой размещения программ, оказавшихся слишком большими и поэтому не помещавшихся в доступной физической памяти. Обычно принималось решение о разделении программы на части, называемые **оверлеями** (overlays). Оверлей 0 обычно запускался первым. После окончания своего выполнения он вызывал следующий оверлей. Некоторые оверлейные системы были очень сложными, позволяющими одновременно находиться в памяти нескольким оверлеям. Оверлеи хранились на диске и по мере необходимости динамически перемещались между памятью и диском средствами операционной системы.

Несмотря на то что фактическая работа по загрузке оверлеев с диска и выгрузке на диск выполнялась системой, делить программы на части должен был программист. Разбиение больших программ на маленькие модули поглощало много времени и было не слишком интересным занятием. Однако такая ситуация продолжалась недолго, так как вскоре кто-то придумал способ поручить всю эту работу компьютеру.

Разработанный метод известен как **виртуальная память** [122]. Основная идея виртуальной памяти заключается в том, что объединенный размер программы, данных и стека может превысить количество доступной физической памяти. Операционная система хранит части программы, использующиеся в настоящий момент, в оперативной памяти, остальные — на диске. Например, программа размером 16 Мбайт сможет работать на машине с 4 Мбайт памяти, если тщательно продумать, какие 4 Мбайт должны храниться в памяти в каждый момент времени. При этом части программы, находящиеся на диске и в памяти, будут меняться местами по мере необходимости.

Виртуальная память может также работать в многозадачной системе при одновременно находящихся в памяти частях многих программ. Когда программа ждет перемещения в память очередной ее части, она находится в состоянии ожидания ввода-вывода и не может работать, поэтому центральный процессор может быть отдан другому процессу тем же самым способом, как в любой другой многозадачной системе.

Страничная организация памяти

Большинство систем виртуальной памяти используют технику, называемую **страничной организацией памяти** (paging), которую мы сейчас опишем. На любом компьютере существует множество адресов в памяти, к которым может обратиться программа. Когда программа использует следующую инструкцию

```
MOV REG.1000
```

она делает это для того, чтобы скопировать содержимое памяти по адресу 1000 в регистр REG (или наоборот, в зависимости от компьютера). Адреса могут формироваться с использованием индексации, базовых регистров, сегментных регистров и другими путями.

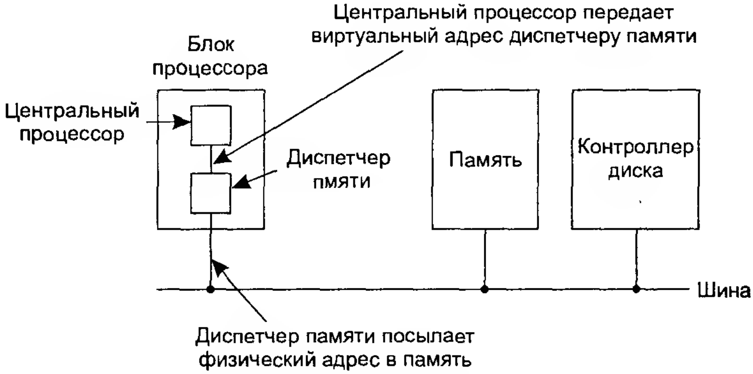


Рис. 4.9. Расположение и функции диспетчера памяти (MMU). Здесь диспетчер памяти показан как часть микросхемы процессора, потому что в наши дни это обычно так и есть. Но логически он мог бы быть отдельной микросхемой, и так было некоторое время назад

Эти программно формируемые адреса, называемые **виртуальными адресами**, формируют **виртуальное адресное пространство**. На компьютерах без виртуальной памяти виртуальные адреса подаются непосредственно на шину памяти и вызывают для чтения или записи слово в физической памяти с тем же самым адресом. Когда используется виртуальная память, виртуальные адреса не передаются напрямую шиной памяти. Вместо этого они передаются **диспетчеру памяти** (MMU — Memory Management Unit), который отображает виртуальные адреса на физические адреса памяти, как продемонстрировано на рис. 4.9.

Очень простой пример того, как работает отображение, приведен на рис. 4.10. Мы рассматриваем компьютер, который может формировать 16-разрядные адреса, от 0 до 64 К. Это виртуальные адреса. Однако у этого компьютера только 32 Кбайт физической памяти, поэтому, хотя программы размером 64 Кбайт могут быть написаны, они не могут целиком быть загружены в память и запущены. Полная копия образа памяти программы размером до 64 Кбайт должна присутствовать на диске, но в таком виде, чтобы ее можно было по мере надобности переносить в память по частям.

Пространство виртуальных адресов разделено на единицы, называемые **страницами**. Соответствующие единицы в физической памяти называются **страничными блоками** (page frame). Страницы и их блоки имеют всегда одинаковый размер. В этом примере они равны 4 Кбайт, но в реальных системах использовались размеры страниц от 512 байт до 64 Кбайт. Имея 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти, мы получаем 16 виртуальных страниц и 8 страничных блоков. Передача данных между ОЗУ и диском всегда происходит в страницах.

Когда программа пытается получить доступ к адресу 0, например, используя команду

```
MOV REG,0
```

виртуальный адрес 0 передается диспетчеру памяти (MMU). Диспетчер памяти видит, что этот виртуальный адрес попадает на страницу 0 (от 0 до 4095), которая отображается страничным блоком 2 (от 8192 до 12287). Диспетчер переводит

виртуальный адрес 0 в физический адрес 8192 и выставляет последний на шину. Память ничего не знает о диспетчере памяти и видит просто запрос на чтение или запись слова по адресу 8192, который и выполняет. Таким образом, диспетчер памяти эффективно отображает все виртуальные адреса между 0 и 4095 на физические адреса от 8192 до 12287.

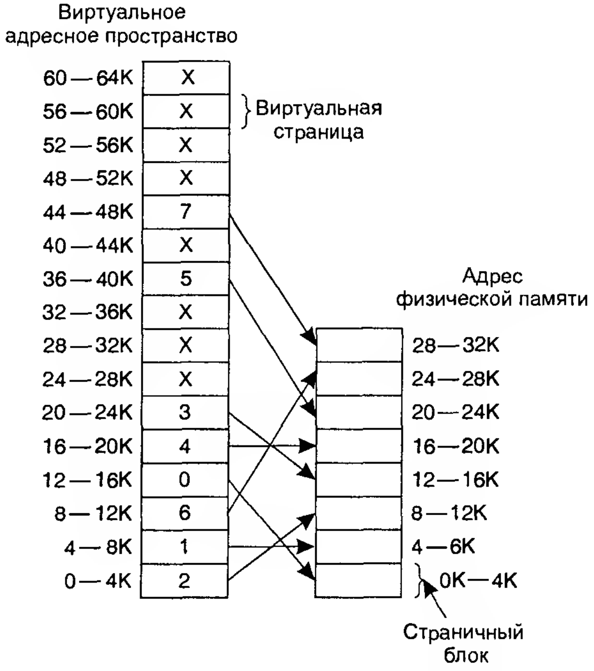


Рис. 4.10. Связь между виртуальными и физическими адресами, получаемая с помощью таблицы страниц

Точно так же инструкция

```
MOV REG,8192
```

преобразуется в команду

```
MOV REG,24576
```

поскольку виртуальный адрес 8192 находится на виртуальной странице 2, а эта страница отображается на физический страничный блок 6 (физические адреса от 24576 до 28671). В качестве третьего примера рассмотрим виртуальный адрес 20500, который адресует 20-й байт от начала виртуальной страницы 5 (виртуальные адреса от 20480 до 24575) и отображается на физический адрес 12288 + 20 = 12308.

Сама по себе возможность отображения 16 виртуальных страниц на любой из восьми страничных блоков с помощью установки соответствующей карты в диспетчере памяти не решает проблемы, заключающейся в том, что размер виртуального адресного пространства больше физической памяти. Так как у нас есть только восемь физических страничных блоков, только восемь виртуальных страниц на

рис. 4.10 воспроизводятся в физической памяти. Другие страницы, обозначенные на рисунке крестиками, не отображаются. В фактическом аппаратном обеспечении страницы, физически присутствующие в памяти, отслеживаются с помощью **бита присутствия/отсутствия**.

Что происходит, если программа пытается воспользоваться неотображаемой страницей, например, с помощью инструкции

```
MOV REG,32780
```

которая обращается к байту 12 на виртуальной странице 8 (начинающейся с адреса 32768)? Диспетчер памяти замечает, что страница не отображается (обозначена крестиком на рисунке), и инициирует прерывание центрального процессора, передающее управление операционной системе. Такое прерывание называется **ошибкой из-за отсутствия страницы** или **страничным прерыванием** (page fault). Операционная система выбирает малоиспользуемый страничный блок и записывает его содержимое на диск. Затем она считывает с диска страницу, на которую произошла ссылка, в только что освободившийся блок, изменяет карту отображения и запускает заново прерванную команду.

Например, если операционная система решает удалить из оперативной памяти страничный блок 1, она загружает виртуальную страницу 8 по физическому адресу 4 К и производит два изменения в карте диспетчера памяти. Во-первых, отмечается содержимое виртуальной страницы 1 как неотображаемое для того, чтобы перехватывать в будущем любые попытки обращения к виртуальным адресам между 4 К и 8 К. Затем заменяется крест в записи для виртуальной страницы 8 на номер 1, так что когда прерванная команда будет выполняться заново, она отобразит виртуальный адрес 32780 на физический адрес 4108.

Теперь рассмотрим диспетчер памяти изнутри, чтобы увидеть, как он работает, и понять, почему мы выбрали размер страницы, являющийся степенью числа 2. На рис. 4.11 представлен пример виртуального адреса 8196 (0010000000000100 в двоичном виде), который отображается с использованием карты диспетчера памяти на рис. 4.10. Входящий 16-разрядный виртуальный адрес разделяется на 4-разрядный номер страницы и 12 битов смещения. При 4 битах под номер страницы в нашей системе может существовать 16 страниц, а с 12 битами смещения мы можем адресоваться ко всем 4096 байтам внутри страницы.

Номер страницы используется в качестве индекса в **таблице страниц**, выдающей номер страничного блока, соответствующего виртуальной странице. Если бит *Присутствия/отсутствия* равен 0, управление переходит к операционной системе. Если этот бит равен 1, то номер страничного блока, найденный в таблице страниц, записывается в три старших бита выходного регистра, а 12 битов смещения копируются без изменения из входящего виртуального адреса. Все вместе они составляют 15-разрядный физический адрес. Затем выходной регистр помещается на шину памяти как адрес физической памяти.

Таблицы страниц

В простейшем случае отображение виртуальных адресов на физические происходит так, как мы только что описали. Виртуальный адрес делится на номер виртуальной страницы (старшие биты) и сдвиг (младшие биты). Например, при 16-разряд-

ных адресах и размере страницы 4 Кбайт старшие 4 бита могут указывать одну из 16 виртуальных страниц, а нижние 12 бит могут определять байт смещения (от 0 до 4095) внутри выбранной страницы. Однако разбиение страницы на 3, 5 или какое-нибудь другое число битов также возможно. Разные части подразумевают различные размеры страниц.

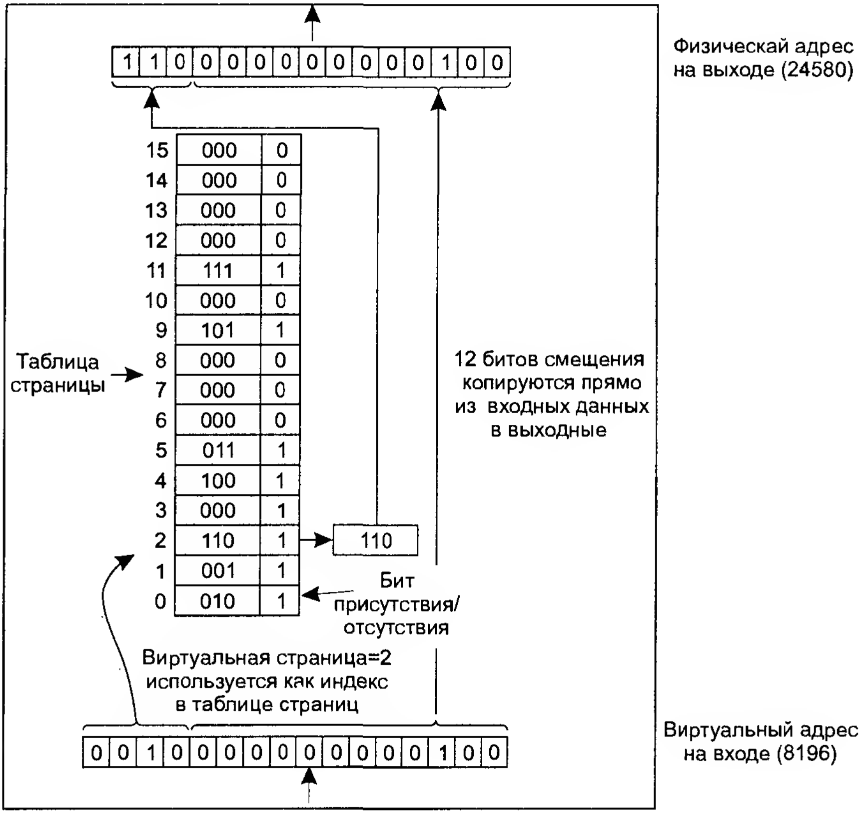


Рис. 4.11. Внутренняя операция диспетчера памяти в системе с шестнадцатью страницами размером 4 Кбайт

Номер виртуальной страницы используется как индекс в таблице страниц для поиска записи этой страницы. По записи в таблице страниц находится номер физического блока страницы (если это имеет место). Данный номер присоединяется к старшим разрядам числа смещения, замещая номер виртуальной страницы и тем самым формируя физический адрес, который может быть послан в память.

Назначение таблицы страниц заключается в отображении виртуальных страниц на страничные блоки. Говоря математически, таблица страниц — это функция, имеющая в качестве аргумента номер виртуальной страницы и получающая в результате номер физического блока. Используя результат действия этой функции, поле виртуальной страницы в виртуальном адресе может быть заменено полем страничного блока, таким образом, формируется физический адрес.

Несмотря на столь простое описание, нам придется столкнуться с двумя важными проблемами:

1. Таблица страниц может быть слишком большой.
2. Отображение должно быть быстрым.

Первый пункт следует из того факта, что современные компьютеры используют по крайней мере 32-разрядные виртуальные адреса. При размере страницы, скажем, 4 Кбайт, 32-разрядное адресное пространство будет состоять из одного миллиона страниц, а 64-разрядное адресное пространство будет включать в себя намного больше страниц, чем то количество, с которым вы захотите иметь дело. При одном миллионе страниц в виртуальном адресном пространстве таблица страниц должна состоять из одного миллиона записей. И помните, что каждый процесс нуждается в своей собственной таблице страниц (потому что у него есть свое собственное виртуальное адресное пространство).

Второй пункт — это вывод из того факта, что преобразование виртуальных адресов в физические должно быть выполнено для каждого обращения к ячейке памяти. Типичная команда процессора включает в себя слово-команду и часто также операнд памяти. В результате необходимо сделать 1, 2 или иногда больше обращений к таблице страниц за команду. Если выполнение команды занимает, скажем, 4 нс, то поиск в таблице страниц должен быть сделан меньше, чем за 1 нс, чтобы преобразование виртуальных адресов не стало главным узким местом системы.

Потребность в огромном, но при этом быстром страничном отображении накладывает существенные ограничения на способы построения компьютеров. Хотя проблема наиболее серьезно встает для старших моделей семейства, она также появляется и для младших моделей, когда стоимость и соотношение цена/производительность имеют критическое значение. В этом и следующих разделах мы рассмотрим устройство таблицы страниц в деталях и покажем несколько аппаратных решений, которые использовались в реальных компьютерах.

Простейшее конструкторское решение (по крайней мере, концептуально) заключается в поддержании таблицы страниц, состоящей из массива быстрых аппаратных регистров с одной записью для каждой виртуальной страницы, индексированного по номерам виртуальных страниц, как показано на рис. 4.11. Когда процесс запускается, операционная система загружает в регистры таблицу страниц процесса, данные берутся из копии, хранящейся в оперативной памяти. Во время выполнения процесса таблице страниц больше не нужно обращаться к памяти. Преимущество этого метода заключается в его простоте и отсутствии необходимости обращений к памяти во время преобразования адресов. Недостатком является его потенциально высокая стоимость (если таблица страниц велика). Необходимость загрузки полной таблицы в регистры при каждом контекстном переключении наносит ущерб производительности.

Другая крайность заключается в том, что таблица страниц целиком располагается в оперативной памяти. Тогда все необходимое оборудование состоит из одного-единственного регистра, указывающего на начало таблицы страниц. Такая схема позволяет изменять карту памяти при контекстном переключении путем перезагрузки только одного регистра. Конечно, она имеет свой недостаток: во время выполнения каждой инструкции программы требуется одно или несколько обраще-

ний к памяти для чтения записей таблицы страниц. По этой причине данный метод редко используется в своем чистом виде, но ниже мы изучим несколько его разновидностей, имеющих намного более высокую производительность.

Многоуровневые таблицы страниц

Чтобы обойти проблему необходимости постоянного хранения в памяти огромных таблиц страниц, многие компьютеры используют многоуровневую таблицу страниц. Простой пример представлен на рис. 4.12. На рис. 4.12, *а* изображен 32-разрядный виртуальный адрес, который разделен на 10-разрядное поле *PT*1, 10-разрядное поле *PT*2 и 12-разрядное поле *Offset* (смещение). Так как под смещение отведено 12 бит, страницы имеют размер 4 Кбайт, и их всего 2^{20}.

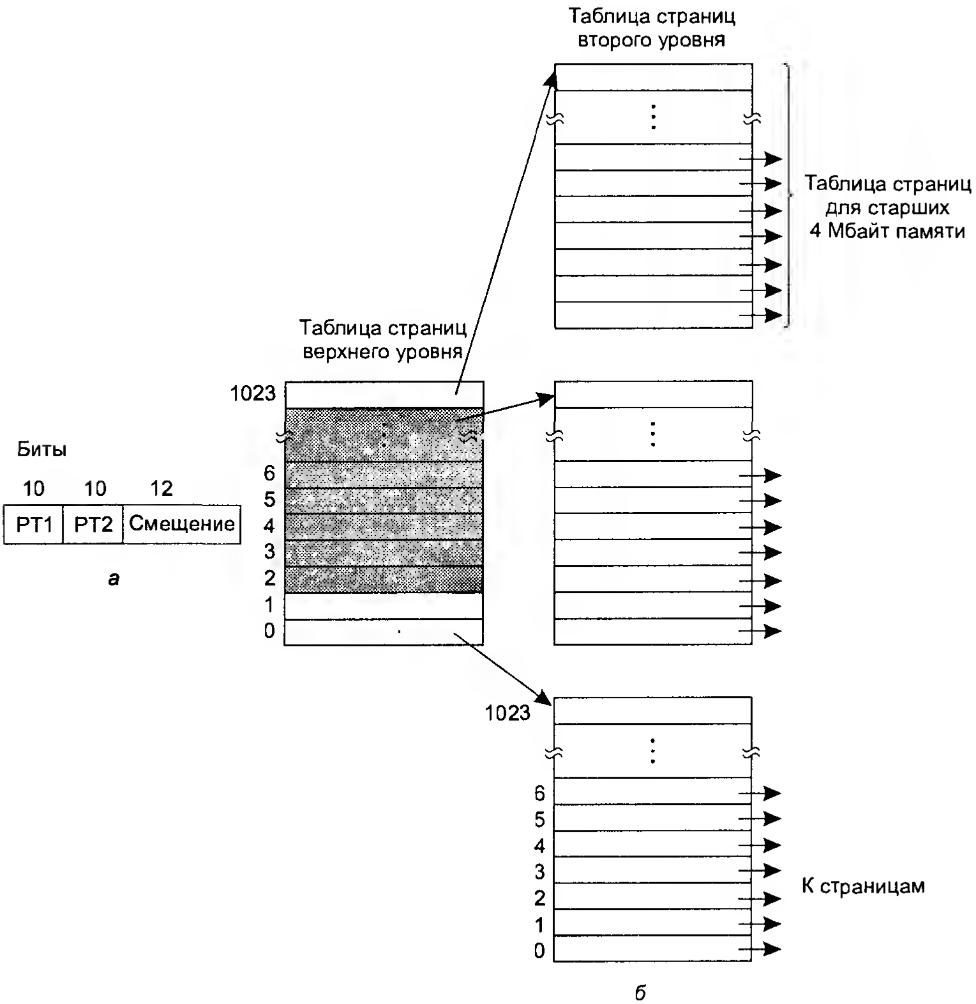


Рис. 4.12. 32-разрядные адреса с полями двух таблиц страниц (*а*); двухуровневая таблица страниц (*б*)

Секрет метода многоуровневой таблицы страниц заключается в том, чтобы избегать постоянного содержания в памяти всех таблиц страниц. В частности, те части, которые не нужны в данный момент, не должны храниться в памяти. Предположим, например, что процессу нужно 12 Мбайт, младшие 4 Мбайт памяти для текста программы, следующие 4 Мбайт для данных и старшие 4 Мбайт для стека. Между верхом данных и низом стека образуется гигантский свободный участок, который не используется.

На рис. 4.12, *б* мы видим, как в данном примере работает двухуровневая таблица страниц. Слева находится таблица страниц верхнего уровня с 1024 записями, соответствующими 10-разрядному полю *PT*1. Когда виртуальный адрес предстает перед диспетчером памяти, он сначала выделяет поле *PT*1 и использует его значение как индекс таблицы верхнего уровня. Каждая из этих 1024 записей представляет 4 М, потому что целое 4-гигабайтное (то есть 32-разрядное) виртуальное адресное пространство было нарезано на куски по 1024 байта.

Запись, место которой определяется по индексу в таблице страниц верхнего уровня, выдает адрес или номер страничного блока таблицы страниц второго уровня. Запись 0 в таблице страниц первого уровня указывает на таблицу страниц для текста программы, запись 1 указывает на таблицу страниц для данных, запись 1023 указывает на таблицу страниц для стека. Другие (заштрихованные) записи не используются. Поле *PT*2 теперь используется как индекс в выбранной таблице второго уровня для поиска номера страничного блока самой страницы.

В качестве примера рассмотрим 32-разрядный адрес 0x00403004 (4 206 596 в десятичном виде), который соответствует байту 12 292 в данных. У этого виртуального адреса *PT*1=1, *PT*2=2 и *Offset*=4. Диспетчер памяти сначала использует поле *PT*1, чтобы по индексу в таблице страниц верхнего уровня получить запись 1, которая соответствует адресам от 4 до 8 М. Затем он воспользуется полем *PT*2, чтобы по индексу из только что найденной таблицы второго уровня извлечь запись 3, которая соответствует адресам от 12 288 до 16 383 внутри своего участка размером 4 М (то есть абсолютным адресам от 4 206 592 до 4 210 687). Эта запись содержит номер физического блока страницы, содержащей виртуальный адрес 0x00403004. Если данная страница не находится в памяти, бит *Присутствия/отсутствия* в записи таблицы страниц будет равен нулю, что приведет к страничному прерыванию. Если страница в памяти, то номер страничного блока, взятый из таблицы страниц второго уровня, присоединяется к смещению (4), создавая физический адрес. Этот адрес выставляется на шину и передается памяти.

Следует отметить одну интересную деталь на рис. 4.12. Хотя адресное пространство содержит больше миллиона страниц, фактически нужны только четыре таблицы: таблица верхнего уровня и таблицы нижнего уровня для памяти от 0 до 4 М, от 4 до 8 М и для верхних 4 М. Битам *Присутствия/отсутствия* для 1021 записи таблицы страниц верхнего уровня присвоено значение 0, что вызовет страничное прерывание при любом обращении к ним. Если это происходит, то операционная система заметит, что процесс пытается обратиться к области памяти, не предполагающей ссылок на нее, и предпримет соответствующее действие, например пошлет ему сигнал или уничтожит его. В описанном выше примере мы выбрали круглые значения для различных величин и выбрали размер поля *PT*1, равный размеру поля *PT*2, но в реальной практике, конечно, возможны другие цифры.

Система двухуровневой таблицы на рис. 4.12 может быть расширена для трех, четырех и больше уровней. Дополнительные уровни дадут большую гибкость, но сомнительно, что следует усложнять систему больше, чем до трех уровней.

Структура элемента таблицы страниц

Теперь от структуры таблиц страниц в целом мы перейдем к деталям отдельного элемента записи таблицы. Точная структура элемента в значительной мере зависит от машины, но виды представленной информации примерно одни и те же. На рис. 4.13 мы привели образец записи в таблице страниц. Ее длина изменяется от компьютера к компьютеру, но 32 бита — это наиболее распространенный размер. Наиболее важным полем является *Номер страничного блока*. Прежде всего, задачей отображения страниц является определение этой величины. За этим полем следует бит *Присутствия/отсутствия*. Если этот бит равен 1, запись имеет силу и может использоваться. Если он равен 0, виртуальная страница, которой соответствует эта запись, в данный момент отсутствует в памяти. Обращение к записи в таблице страниц, у которой этому биту присвоено нулевое значение, приводит к страничному прерыванию.

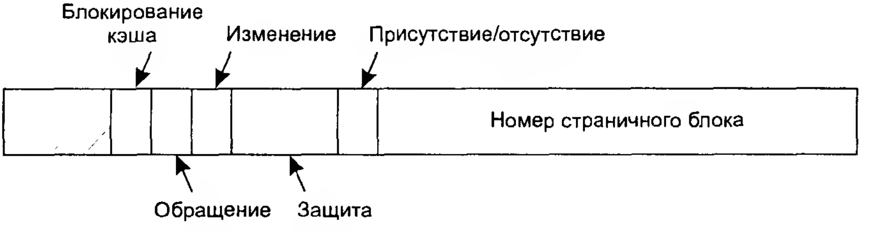


Рис. 4.13. Типичная запись в таблице страниц

Биты *Защиты* говорят о том, какие разрешены виды доступа к этой странице. В простейшей форме это поле содержит один бит, равный 1 для чтения/записи и равный 0 только для чтения. Более сложные схемы имеют три бита, по одному для допуска каждой из операций чтения, записи и выполнения страницы.

Биты *Изменения* и *Обращения* отслеживают использование страницы. Когда страница записывается, аппаратура автоматически устанавливает бит *Изменение*. Этот бит учитывается, когда операционная система решает освободить страничный блок. Если страница в нем была изменена (то есть она «грязная»), то ее новая версия должна быть переписана на диск. Если она не была модифицирована (то есть страница «чистая»), ее можно просто удалить из памяти, так как все еще действительна копия на диске. Этот бит иногда называют **грязным битом**, так как он отражает состояние страницы.

Бит *Обращения* устанавливается всякий раз, когда происходит обращение к странице для чтения или записи. Его значение помогает операционной системе при выборе страницы для удаления из памяти, когда случается ошибка из-за отсутствия страницы. Страницы, не использующиеся в данный момент, являются лучшими кандидатами, чем находящиеся в работе. Этот бит играет важную роль в нескольких алгоритмах перемещения страниц, которые мы изучим позже в этой главе.

Наконец, последний бит позволяет запретить кэширование страницы. Это свойство важно для страниц, отображающихся не на память, а на регистры устройств. Если операционная система находится в цикле ожидания ответа от некоторого устройства ввода-вывода, которому была только что дана команда, существенно, чтобы аппаратура продолжала получать слово из устройства, а не использовало старую копию, находящуюся в кэш-памяти. При помощи этого бита кэширование можно отключить. Машины, имеющие отдельное пространство адресов ввода-вывода и не использующие отображения регистров ввода-вывода на память, не нуждаются в этом бите.

Заметим, что адрес места на диске, в котором хранится страница тогда, когда она не находится в памяти, не является частью таблицы страниц. Причина очень проста. Таблица страниц содержит только ту информацию, которая нужна аппаратуре для перевода виртуального адреса в физический. Информация, необходимая операционной системе для обработки страничных прерываний, хранится в программных таблицах внутри операционной системы. Аппаратуре она не нужна.

Буферы быстрого преобразования адреса (TLB)

В большинстве схем со страничной организацией памяти таблицы страниц хранятся в памяти из-за их значительного размера. Потенциально такое устройство оказывает колоссальное влияние на производительность. Рассмотрим, например, команду процессора, копирующую содержимое одного регистра в другой. В отсутствие страничной организации памяти эта команда приводит только к одному обращению к памяти для выборки самой команды. Если же память организована постранично, то будут необходимы дополнительные ссылки для доступа к таблице страниц. Так как скорость выполнения команд в основном ограничена скоростью, с которой центральный процессор выбирает команды и данные из памяти, необходимость двух обращений к таблице страниц на одну ссылку к памяти уменьшает производительность на 2/3. При таких условиях никто не стал бы использовать этот метод.

Разработчики компьютеров многие годы размышляли об этой проблеме и в результате придумали решение. Оно основано на наблюдении, что большинство программ склонно делать огромное количество обращений к небольшому количеству страниц, а не наоборот. Таким образом, в таблице страниц только малая доля записей читается интенсивно, остальная часть едва ли вообще используется.

В результате принятого решения компьютер снабжается небольшим аппаратным устройством, служащим для отображения виртуальных адресов в физические без прохода по таблице страниц. Это устройство, называемое **буфером быстрого преобразования адреса** (TLB — Translation Lookaside Buffer) или иногда **ассоциативной памятью**, продемонстрировано в табл. 4.1. Оно обычно находится внутри диспетчера памяти и состоит из нескольких записей. В этом примере их восемь, но фактически записей редко бывает больше 64. Каждая запись содержит информацию об одной странице, а именно: номер виртуальной страницы, бит, устанавливаемый при изменении страницы, код защиты (разрешения на чтение/запись/выполнение) и номер физического страничного блока, в котором расположена эта страница. Эти поля однозначно соответствуют полям в таблице страниц. Еще один бит служит признаком того, действительна ли запись (то есть используется ли она в данный момент) или нет.

Таблица 4.1. Буфер быстрого преобразования памяти для увеличения скорости страничной подкачки

Действительный	Виртуальная страница	Изменение	Защита	Страничный фрейм
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Пример, который мог бы сформировать TLB-буфер, изображенный на рис. 4.14, — это циклический процесс, располагающийся в виртуальных страницах 19, 20 и 21, поэтому эти записи в буфере на рис. 4.14 имеют защитные коды для чтения и выполнения. Основные данные, используемые в настоящее время (скажем, обрабатываемый массив), находятся в страницах 129 и 130. Страница 140 содержит индексы, требующиеся для вычислений массива. И наконец, в страницах 860 и 861 находится стек.

Теперь рассмотрим, как же функционирует буфер быстрого преобразования адреса (TLB). Когда виртуальный адрес представляется диспетчером памяти для отображения, аппаратура сначала убеждается в том, что номер его виртуальной страницы присутствует в буфере TLB путем сравнения адреса со всеми записями одновременно (то есть параллельно). Если найдено имеющее силу совпадение и обращение не нарушает биты защиты, страничный блок берется прямо из буфера TLB, без перехода к таблице страниц. Если номер виртуальной страницы присутствует в буфере TLB, но инструкция пытается записать что-то на страницу, доступную только для чтения, формируется ошибка защиты точно так же, как это происходило бы из самой таблицы страниц.

Интересная ситуация получается, если номер виртуальной страницы не находится в буфере быстрого преобразования адреса. Диспетчер памяти обнаруживает отсутствие страницы и выполняет обычный поиск в таблице страниц. Затем он удаляет одну из записей из буфера TLB и заменяет ее только что найденной записью из таблицы страниц. Таким образом, если эта страница снова вскоре будет затребована, во второй раз поиск окажется успешным, а не неудачным. Когда запись удаляется из буфера быстрого преобразования адреса, бит изменения копируется в запись таблицы страниц в памяти. Другие величины уже находятся там. Когда буфер TLB загружается из таблицы страниц, все поля берутся из памяти.

Программное управление буфером TLB

До сих пор мы предполагали, что каждая машина со страничной виртуальной памятью имеет таблицы страниц, распознаваемые аппаратным обеспечением и буфером быстрого преобразования адреса. При таком устройстве управление буфером TLB и обработка его ошибок выполняются полностью аппаратурой диспетчера памяти (MMU). Передача управления операционной системе происходит только тогда, когда страница отсутствует в памяти.

В прошлом это допущение было справедливо. Однако многие современные RISC-компьютеры, включая машины SPARC, MIPS, Alpha и HP PA, выполняют почти все страничное управление программно. На этих машинах записи буферы TLB явно загружаются операционной системой. Когда происходит неудачный поиск в буфере TLB, диспетчер памяти вместо того, чтобы переходить к таблице страниц для поиска и выбора необходимой страницы, формирует ошибку буфера TLB и передает проблему в руки операционной системы. Система должна найти страницу, удалить запись из буфера TLB, ввести новую запись и перезапустить прерванную инструкцию. И, конечно, все это должно быть сделано при помощи небольшого числа команд, потому что неудачи поиска в буфере быстрого преобразования адреса происходят намного чаще, чем ошибки из-за отсутствия страниц.

Достаточно удивительно то, что если буфер TLB имеет небольшой размер (скажем, 64 записи) для понижения частоты неудачных пропусков, программное управление буфером, оказывается, является приемлемо результативным. Главная выгода здесь заключается в намного более простом устройстве диспетчера памяти, что освобождает достаточное количество пространства в микросхеме процессора для кэша и других устройств, способных повысить производительность. Программное управление буфером быстрого преобразования адреса обсуждается в [333].

Для улучшения производительности на машинах, программно управляющих буфером TLB, разрабатывались различные стратегии поведения. Один подход состоит в попытке уменьшить как частоту неудачного поиска в буфере, так и его стоимость, когда он все-таки случается [19]. Чтобы уменьшить вероятность неудачного поиска в буфере TLB, иногда операционная система может интуитивно вычислить, какие страницы, возможно, будут использоваться следующими и предварительно загрузить записи для них в буфер TLB. Например, когда клиентский процесс посылает сообщение серверному процессу на той же самой машине, очень вероятно, что сервер вскоре должен будет начать работу. Зная это, система может также проверить, где находятся страницы кода сервера, данных и стека, пока прерывание обрабатывается, чтобы осуществить вызов send, и преобразовать их адреса из виртуальных в физические до того, как они смогут стать причиной ошибки TLB-буфера.

Обычный путь обработки неудачного поиска в буфере TLB, аппаратно или программно — это переход в таблицу страниц и выполнение операции индексации, чтобы определить место страницы, к которой происходит обращение. При осуществлении этого поиска программно возникает проблема, заключающаяся в том, что страницы, содержащиеся в таблице страниц, могут отсутствовать в буфере быстрого преобразования адреса, что вызовет дополнительные ошибки буфера TLB во время обработки. Их количество можно уменьшить, поддерживая большой (например, 4 Кбайт) программный кэш записей буфера TLB с фиксированным расположением в памяти, чьи страницы всегда хранятся в буфере TLB. Если сначала проверять программный кэш, операционная система может в значительной степени снизить количество неудачных поисков в буфере TLB.

Инвертированные таблицы страниц

Традиционные таблицы страниц, тип которых мы описывали до сих пор, требуют по одной записи на каждую виртуальную страницу, так как они индексируются по номеру этой страницы. Если адресное пространство состоит из 2^{32} байт с размером

страницы 4096 байт, тогда в таблице страниц должно быть больше миллиона записей. При этом таблица страниц будет занимать минимум 4 Мбайт. В достаточно больших системах это, вероятно, выполнимо.

Однако поскольку 64-разрядные компьютеры встречаются все чаще, ситуация радикально меняется. Если теперь адресное пространство увеличилось до 2^{64} байт с размером страницы 4 Кбайт, нам требуется таблица страниц с 2^{52} записями. Если каждая запись равна 8 байтам, таблица займет больше 30 Тбайт. Выделение 30 Тбайт только для таблицы страниц нереально сейчас и не будет реальным когда-либо в будущем. Следовательно, для 64-разрядного страничного виртуального пространства необходимо другое решение.

Одним из таких решений является **инвертированная таблица страниц**. В этой модели таблица содержит по одной записи на страничный блок в реальной памяти, а не на страницу в виртуальном адресном пространстве. Например, при 64-разрядных виртуальных адресах, размере страниц 4 Кбайт и 256 Мбайт оперативной памяти инвертированная таблица страниц потребует всего лишь 65 536 записей. Каждая запись отслеживает, что (процесс, виртуальная страница) расположено в данном страничном блоке.

Хотя инвертированные таблицы страниц экономят значительное количество места, по крайней мере, когда виртуальное адресное пространство намного больше, чем физическая память, они имеют серьезный недостаток: перевод виртуального адреса в физический становится намного сложнее. Когда процесс *n* обращается к виртуальной странице *p*, аппаратное обеспечение не может больше найти физическую страницу, используя номер *p* в качестве индекса в таблице страниц. Вместо этого оно должно производить поиск записи (*n*, *p*) во всей инвертированной таблице страниц. Более того, этот поиск должен выполняться при каждом обращении к памяти, а не только при страничном прерывании. Операция поиска в таблице размером 64 К при каждой ссылке к памяти вовсе не увеличит скорость вашей машины.

Выйти из этого затруднительного положения можно, используя буфер быстрого преобразования адреса (TLB). Если буфер TLB может содержать все часто используемые страницы, трансляция адреса будет происходить так же быстро, как и с обычными таблицами страниц. Но при неудачном поиске в буфере TLB поиск в инвертированной таблице страниц должен выполняться программно. Один из возможных способов усовершенствовать его — поддерживать хэш-таблицу виртуальных адресов. Все виртуальные страницы, находящиеся в данный момент в памяти и имеющие одинаковое значение хэш-функции, сцепляются друг с другом, как показано на рис. 4.14. Если хэш-таблица состоит из такого же количества ячеек, сколько в машине физических страниц, средняя цепочка будет длиной только в одну запись, что значительно увеличит скорость отображения адресов. Как только найден номер страничного блока, новая пара (виртуальная, физическая) помещается в буфер TLB.

Инвертированные таблицы страниц в настоящее время используются на некоторых рабочих станциях компаний IBM и Hewlett-Packard и будут встречаться все чаще, так как 64-разрядные машины получают все более широкое распространение. Некоторые другие методы управления виртуальной памятью большого размера можно найти в [159, 320, 321].

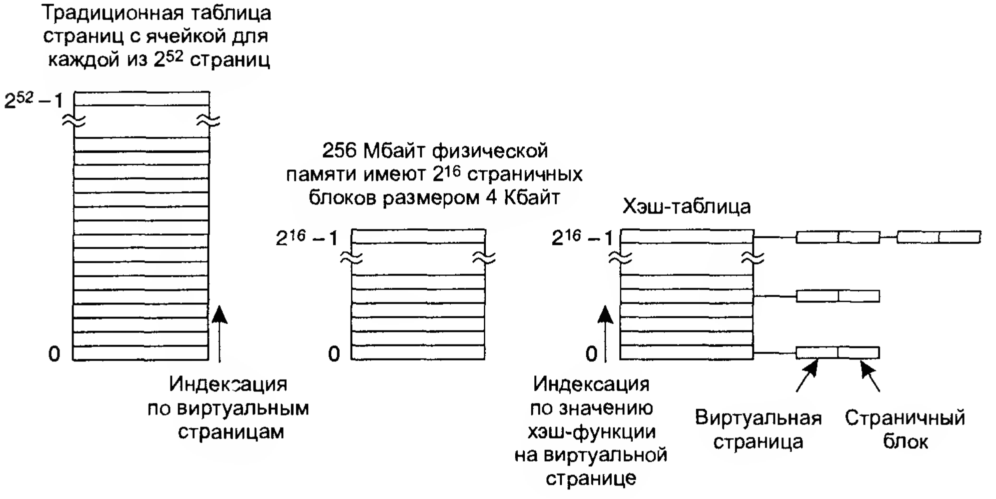


Рис. 4.14. Сравнение традиционной таблицы страниц с инвертированной

Алгоритмы замещения страниц

Когда происходит страничное прерывание, операционная система должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, которую нужно перенести в память. Если удаляемая страница была изменена за время своего присутствия в памяти, ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована (например, она содержит текст программы), копия на диске уже является самой новой и ее не надо переписывать. Тогда страница, которую нужно прочитать, просто считывается поверх выгружаемой страницы.

Хотя в принципе можно при каждом страничном прерывании выбирать случайную страницу для удаления из памяти, производительность системы заметно повышается, когда предпочтение отдается редко используемой странице. Если выгружается страница, обращения к которой происходят часто, велика вероятность, что вскоре опять потребуется ее возврат в память, что даст в результате дополнительные издержки. Теме разработки алгоритмов замены страницы было посвящено много работ, как теоретических, так и экспериментальных. Ниже мы опишем некоторые из наиболее важных алгоритмов.

Следует отметить, что проблема «страничного обмена» также встает и в других областях конструирования компьютеров. Например, у большинства компьютеров есть один или несколько кэшей, состоящих из используемых в последнее время 32-байтовых или 64-байтовых блоков памяти. Когда кэш заполнен, необходимо выбрать некоторые блоки для удаления. Эта проблема практически аналогична замещению страниц лишь с одной разницей, заключающейся в меньшем масштабе времени (операция должна быть выполнена за несколько наносекунд, а не миллисекунд, как для замены страниц). Причиной для более короткого промежутка времени является то, что неудачный поиск блока в кэше обрабатывается из основной

памяти, в которой не тратится время на поиск нужного цилиндра диска и нет задержки из-за его вращения.

Второй пример встречается на web-серверах. Сервер может хранить определенное количество часто используемых web-страниц в своей кэш-памяти. Однако когда кэш-память заполняется целиком и происходит обращение к новой странице, должно приниматься решение о том, какую из страниц выгружать. Здесь применимы те же рассуждения, что и для страниц в виртуальной памяти, с той разницей, что web-страницы никогда не изменяются в кэше, поэтому для них всегда есть свежая копия на диске. В системе виртуальной памяти страницы в оперативной памяти могут быть «чистыми» или «грязными».

Оптимальный алгоритм

Наилучший из возможных алгоритмов замещения страниц легко описать, но невозможно осуществить. Он действует так. В тот момент, когда происходит страничное прерывание, в памяти находится некоторый набор страниц. К одной из этих страниц будет обращаться следующая команда процессора (к странице, содержащей требуемую команду). На другие страницы, возможно, не будет ссылок в течение следующих 10, 100 или даже 1000 команд. Каждая страница может быть помечена количеством команд, которые будут выполняться перед первым обращением к этой странице.

Оптимальный страничный алгоритм просто сообщает, что должна быть выгружена страница с наибольшей меткой. Если одна страница не будет использоваться в течение 8 млн команд, а другая — в течение 6 млн инструкций, удаление первой отодвинет в будущее на возможно максимальный срок страничное прерывание, которое вернет ее назад. Компьютеры, подобно людям, пытаются отложить неприятные события настолько, насколько это возможно.

С этим алгоритмом связана только одна проблема: он невыполним. В момент страничного прерывания операционная система не имеет возможности узнать, когда произойдет следующее обращение к каждой странице. (Мы рассматривали аналогичную ситуацию раньше, когда обсуждали алгоритм планирования «кратчайшая задача — первая»: как система может сказать, какая из задач самая короткая?) Тем не менее, выполняя программу на модели и следя за всеми обращениями к страницам, оптимальную замену можно осуществить при *втором* запуске, используя информацию о ссылках на страницы, собранную во время *первого* запуска.

В этом случае можно сравнивать производительность реализуемых алгоритмов с наилучшим. Если операционная система добивается производительности, скажем, всего на один процент ниже, чем при работе оптимального алгоритма, усилия, потраченные на поиск лучшего алгоритма, повысят продуктивность схемы максимум на 1 %.

Чтобы избежать возможных недоразумений, следует прояснить, что полученный протокол обращений к страницам относится только к одной хорошо спланированной программе и, кроме того, к определенным входным данным. Таким образом, алгоритм замещения страниц, выведенный из него, будет характерен только для этой программы с именно этими входными данными. Хотя такой метод полезен для оценки алгоритмов замещения страниц, он не используется в практических системах. Ниже мы изучим алгоритмы, которые *являются* применимыми в реальных системах.

Алгоритм NRU — не использовавшаяся в последнее время страница

Чтобы дать возможность операционной системе собирать полезные статистические данные о том, какие страницы используются, а какие — нет, большинство компьютеров с виртуальной памятью поддерживают два статусных бита, связанных с каждой страницей. Бит *R* (Referenced — обращения) устанавливается всякий раз, когда происходит обращение к странице (чтение или запись). Бит *M* (Modified — изменение) устанавливается, когда страница записывается (то есть изменяется). Биты содержатся в каждом элементе таблицы страниц, как показано на рис. 4.13. Важно реализовать обновление этих битов при каждом обращении к памяти, поэтому необходимо, чтобы они задавались аппаратно. Если однажды бит был установлен в 1, то он остается равным 1 до тех пор, пока операционная система программно не вернет его в состояние 0.

Если аппаратное обеспечение не поддерживает эти биты, их можно смоделировать следующим образом. Когда процесс запускается, все его записи в таблице страниц помечаются как отсутствующие в памяти. Как только происходит обращение к странице, происходит страничное прерывание. Затем операционная система устанавливает бит *R* (в своих внутренних таблицах); изменяет запись в таблице страниц, чтобы она указывала на корректную страницу с режимом READ ONLY (только для чтения), и перезапускает команду. Если страница позднее записывается, происходит другое страничное прерывание, позволяющее операционной системе установить бит *M* и изменить состояние страницы на READ/WRITE (чтение/запись).

Биты *R* и *M* могут использоваться для построения простого алгоритма замещения страниц, описанного ниже. Когда процесс запускается, оба страничных бита для всех его страниц операционной системой установлены на 0. Периодически (например, при каждом прерывании по таймеру) бит *R* очищается, чтобы отличить страницы, к которым давно не происходило обращения от тех, на которые были ссылки.

Когда возникает страничное прерывание, операционная система проверяет все страницы и делит их на четыре категории на основании текущих значений битов *R* и *M*:

Класс 0: не было обращений и изменений.
Класс 1: не было обращений, страница изменена.
Класс 2: было обращение, страница не изменена.
Класс 3: произошло и обращение, и изменение.

Хотя класс 1 на первый взгляд кажется невозможным, такое случается, когда у страницы из класса 3 бит *R* сбрасывается во время прерывания по таймеру. Прерывания по таймеру не стирают бит *M*, потому что эта информация необходима для того, чтобы знать, нужно ли переписывать страницу на диске или нет. Поэтому если бит *R* устанавливается на ноль, а *M* остается нетронутым, страница попадает в класс 1.

Алгоритм **NRU** (Not Recently Used — не использовавшийся в последнее время) удаляет страницу с помощью случайного поиска в непустом классе с наименьшим номером. В этом алгоритме подразумевается, что лучше выгрузить измененную страницу, к которой не было обращений по крайней мере в течение одного тика

системных часов (обычно 20 мс), чем стереть часто используемую страницу. Привлекательность алгоритма NRU заключается в том, что он легок для понимания, умеренно сложен в реализации и дает производительность, которая, конечно, не оптимальна, но может вполне оказаться достаточной.

Алгоритм FIFO — первым прибыл — первым обслужен

Другим требующим небольших издержек алгоритмом является **FIFO** (First-In, First-Out — «первым прибыл — первым обслужен»). Чтобы проиллюстрировать его работу, рассмотрим универсам, на полках которого можно выставить ровно k различных продуктов. Он предлагает новую удобную пищу: растворимый, глубоко замороженный, экологически чистый йогурт, который можно мгновенно приготовить в микроволновой печи. Покупатели тут же обратили внимание на этот продукт, поэтому наш ограниченный в размерах супермаркет, для того чтобы продавать йогурт, должен избавиться от одного из старых товаров.

Один из вариантов состоит в том, чтобы найти продукт, который супермаркет продает дольше всего (то есть что-нибудь, что начали реализовывать 120 лет назад), и освободить от него магазин на том основании, что им никто больше не интересуется. В действительности супермаркет хранит перечень всех продаваемых в данный момент товаров, упорядоченный по времени их появления. Каждый новый продукт помещается в конец перечня, а из начала списка удаляется одно старое наименование.

Ту же самую идею можно применить в качестве алгоритма замещения страниц. Операционная система поддерживает список всех страниц, находящихся в данный момент в памяти, в котором первая страница является старейшей, а страницы в хвосте списка попали в него совсем недавно. Когда происходит страничное прерывание, выгружается из памяти страница в голове списка, а новая страница добавляется в его конец. Если алгоритм FIFO использовать в магазине, то он может удалить воск для усов, но также может удалить и муку, соль или масло. Применительно к компьютерам возникает та же проблема. По этой причине алгоритм FIFO редко используется в своей исходной форме.

Алгоритм «вторая попытка»

В простейшем варианте алгоритма FIFO, который позволяет избежать проблемы вытеснения из памяти часто используемых страниц, у самой старейшей страницы изучается бит R. Если он равен 0, страница не только находится в памяти долго, она вдобавок еще и не используется, поэтому немедленно заменяется новой. Если же бит R равен 1, то ему присваивается значение 0, страница переносится в конец списка, а время ее загрузки обновляется, то есть считается, что страница только что попала в память. Затем процедура продолжается.

Работа этого алгоритма, называемого **«второй попыткой»** (second chance), показана на рис. 4.15, *а*. Здесь изображены страницы от A до H, хранящиеся в связанном списке и отсортированные по времени их поступления в память. Числа над страницами обозначают их время загрузки в память.

Предположим, что в момент времени 20 происходит страничное прерывание. Самой старшей страницей является страница *А*, она была загружена в память во время 0, когда начал работу процесс. Если бит *R* страницы *А* равен 0, она выгружается из памяти или путем записи на диск (если страница «грязная»), или просто удаляется (если она «чистая»). Во втором случае, если бит *R* равен 1, страница *А* передвигается в конец списка, а ее «загрузочное время» принимает текущее значение (20). При этом бит *R* очищается[1]. Поиск подходящей страницы продолжается; следующей проверяется страница *В*.

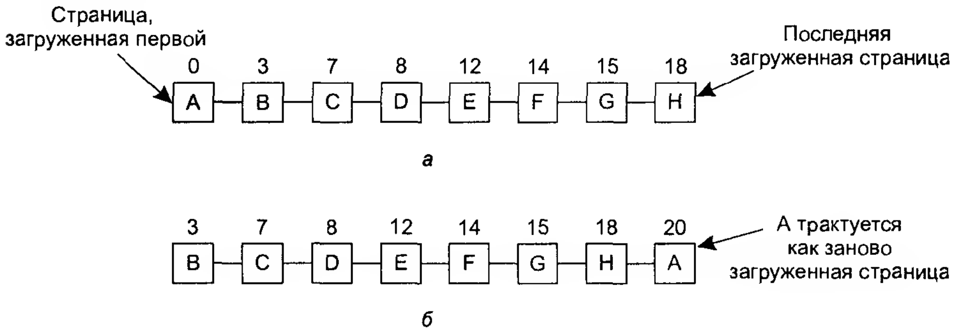


Рис. 4.15. Действие алгоритма «вторая попытка»: страницы, отсортированные в порядке очереди (FIFO) (*а*); список страниц, если страничное прерывание произошло во время 20, а страница *А* имеет бит *R*, равный 0 (*б*)

Алгоритм «вторая попытка» ищет в списке самую старую страницу, к которой не было обращений в предыдущем временном интервале. Если же происходили ссылки на все страницы, то «вторая попытка» превращается в обычный алгоритм FIFO. Представьте, что у всех страниц на рис. 4.15, *а* бит *R* равен 1. Одну за другой передвигает операционная система страницы в конец списка, очищая бит *R* каждый раз, когда она перемещает страницу в хвост. Наконец, она вернется к странице *А*, но теперь уже ее биту *R* присвоено значение 0. В этот момент страница *А* выгружается из памяти. Таким образом, алгоритм всегда успешно завершает свою работу.

Алгоритм «часы»

Хотя алгоритм «вторая попытка» является корректным, он слишком неэффективен, потому что постоянно передвигает страницы по списку. Поэтому лучше хранить все страничные блоки в кольцевом списке в форме часов, как показано на рис. 4.16. Стрелка указывает на старейшую страницу.

Когда происходит страничное прерывание, проверяется та страница, на которую направлена стрелка. Если ее бит *R* равен 0, страница выгружается, на ее место в часовой круг встает новая страница, а стрелка сдвигается вперед на одну позицию. Если бит *R* равен 1, то он сбрасывается, стрелка перемещается к следующей странице. Этот процесс повторяется до тех пор, пока не находится та страница,

[1] То есть устанавливается на 0. — *Примеч. перев.*

у которой бит $R = 0$. Неудивительно, что этот алгоритм называется **«часы»**. Он отличается от алгоритма «вторая попытка» только своей реализацией.

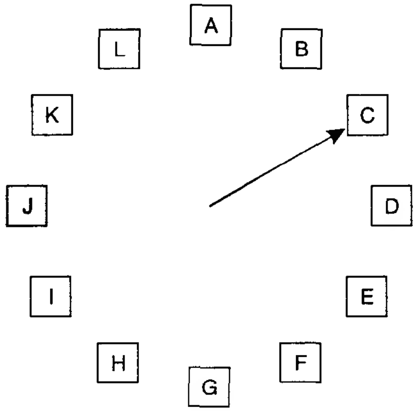


Рис. 4.16. Алгоритм замещения страниц «часы»

Алгоритм LRU — страница, не использовавшаяся дольше всего

В основе этой неплохой аппроксимации оптимального алгоритма лежит наблюдение, что страницы, к которым происходило многократное обращение в нескольких последних командах, вероятно, также будут часто использоваться в следующих инструкциях. И наоборот, можно полагать, что страницы, к которым ранее не возникало обращений, не будут употребляться в течение долгого времени. Эта идея привела к следующему реализуемому алгоритму: когда происходит страничное прерывание, выгружается из памяти страница, которая не использовалась дольше всего. Такая стратегия замещения страниц называется **LRU** (Least Recently Used — «менее недавно», то есть наиболее давно использовавшаяся страница).

Хотя алгоритм LRU теоретически реализуем, он не является дешевым. Для полного осуществления алгоритма LRU необходимо поддерживать связный список всех содержащихся в памяти страниц, где последняя использовавшаяся страница находится в начале списка, а та, к которой дольше всего не было обращений, — в конце. Сложность заключается в том, что список должен обновляться при каждом обращении к памяти. Поиск страницы, ее удаление, а затем вставка в начало списка — это операции, поглощающие очень много времени, даже если они выполняются аппаратно (если предположить, что необходимое оборудование можно сконструировать).

Однако существуют другие способы реализации алгоритма LRU с помощью специального оборудования. Для первого метода требуется оснащение компьютера 64-разрядным аппаратным счетчиком C, который автоматически возрастает после каждой команды. Кроме того, каждая запись в таблице страниц должна иметь поле, достаточно большое для хранения значения счетчика. После каждого обращения к памяти текущая величина счетчика C запоминается в записи таблицы, соответствующей той странице, к которой произошла ссылка. А если возникает стра-

ничное прерывание, операционная система проверяет все значения счетчиков в таблице страниц и ищет наименьшее. Эта страница является не использовавшейся дольше всего.

Теперь рассмотрим второй вариант аппаратной реализации алгоритма LRU. На машине с n страничными блоками оборудование LRU может поддерживать матрицу $n \times n$ бит, изначально равных нулю. Всякий раз, когда происходит обращение к страничному блоку k, аппаратура сначала присваивает всем битам строки k значение 1, затем приравнивает к нулю все биты столбца k. В любой момент времени строка, двоичное значение которой наименьшее, является не использовавшейся дольше всего. Работа этого алгоритма продемонстрирована на рис. 4.17, где рассматриваются четыре страничных блока и следующий порядок обращения к страницам:

0 1 2 3 2 1 0 3 2 3

После ссылки на страницу 0 мы получаем ситуацию, показанную на рис. 4.17, *а*; после обращения к странице 1 — рис. 4.17, *б* и т. д.

а

Страница	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

б

Страница	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

в

Страница	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

г

Страница	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

д

Страница	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

е

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

ж

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

з

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

и

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

к

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

Рис. 4.17. Алгоритм LRU, использующий матрицу. Обращения к страницам происходят в порядке: 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

Программное моделирование алгоритма LRU

Хотя оба описанных выше алгоритма LRU в принципе осуществимы, очень мало (если вообще такие есть) машин оснащено подобным оборудованием, поэтому разработчики операционных систем для компьютеров, не имеющих такой аппаратуры, редко используют эти алгоритмы. Вместо этого требуется программно реализуемое решение. Одна из разновидностей схемы LRU называется алгоритмом **NFU** (Not Frequently Used — редко использовавшаяся страница). Для него необходим программный счетчик, связанный с каждой страницей в памяти, изначально равный нулю. Во время каждого прерывания по таймеру операционная система исследует

все страницы в памяти. Бит R каждой страницы (он равен 0 или 1) прибавляется к счетчику. В сущности, счетчики пытаются отследить, как часто происходило обращение к каждой странице. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Основная проблема, возникающая при работе с алгоритмом NFU, заключается в том, что он никогда ничего не забывает. Например, в многоходовом компиляторе страницы, которые часто использовались во время первого прохода, могут все еще иметь высокое значение счетчика при более поздних проходах. Фактически, если случается так, что первый проход занимает самое долгое время выполнения из всех, страницы, содержащие программный код для следующих проходов, могут всегда иметь более низкое значение счетчика, чем страницы первого прохода. Следовательно, операционная система удалит полезные страницы вместо тех, которые больше не нужны.

К счастью, небольшая доработка алгоритма NFU делает его способным моделировать алгоритм LRU достаточно хорошо. Изменение состоит из двух частей. Во-первых, каждый счетчик сдвигается вправо на один разряд перед прибавлением бита R. Во-вторых, бит R добавляется в крайний слева, а не в крайний справа бит счетчика.

На рис. 4.18 продемонстрировано, как работает видоизмененный алгоритм, известный под названием **«старение»** (aging). Предположим, что после первого тика часов биты R для страниц от 0 до 5 имеют значения 1, 0, 1, 0, 1, 1 соотвественно (у страницы 0 бит R равен 1, у страницы 1 $R = 0$, у страницы 2 $R = 1$ и т. д.). Другими словами, между тиком 0 и тиком 1 произошло обращение к страницам 0, 2, 4 и 5, их биты R приняли значение 1, остальные сохранили значение 0. После того как шесть соответствующих счетчиков сдвинулись на разряд и бит R занял крайнюю слева позицию, счетчики получили значения, показанные на рис. 4.18, *а*. Остальные четыре колонки рисунка изображают шесть счетчиков после следующих четырех тиков часов.

Когда происходит страничное прерывание, удаляется та страница, чей счетчик имеет наименьшую величину. Ясно, что счетчик страницы, к которой не было обращений, скажем, за четыре тика, будет начинаться с четырех нулей и, таким образом, иметь более низкое значение, чем счетчик страницы, на которую не ссылались в течение только трех тиков часов.

Эта схема отличается от алгоритма LRU в двух случаях. Рассмотрим страницы 3 и 5 на рис. 4.18, *д*. Ни к одной из них не было обращений за последние два тика, к обеим было обращение за предшествующий этому тик. Следуя алгоритму LRU, при удалении страницы из памяти мы должны выбрать одну из этих двух. Проблема в том, что мы не знаем, к какой из них позже произошло обращение в интервале времени между тиками 1 и 2. Записывая только один бит за промежуток времени, мы теряем возможность отличить более ранние от более поздних обращений в этом интервале времени. Все, что мы можем сделать — это выгрузить страницу 3, потому что к странице 5 также обращались двумя тиками раньше, а к странице 3 — нет.

Второе отличие между алгоритмами LRU и «старения» заключается в том, что в последнем счетчик имеет конечное число разрядов, например 8. Предположим, что каждая из двух страниц имеет значение счетчика, равное 0. В данной ситуации мы только случайным образом можем выбрать одну из них. На самом деле

может оказаться, что к одной странице в последний раз обращались 9 тиков назад, а к другой — 1000. И мы не имеем возможности увидеть это. На практике, однако, обычно достаточно 8 битов при тике системных часов около 20 мс. Если к странице не обращались в течение 160 мс, очень вероятно, что она не важна.

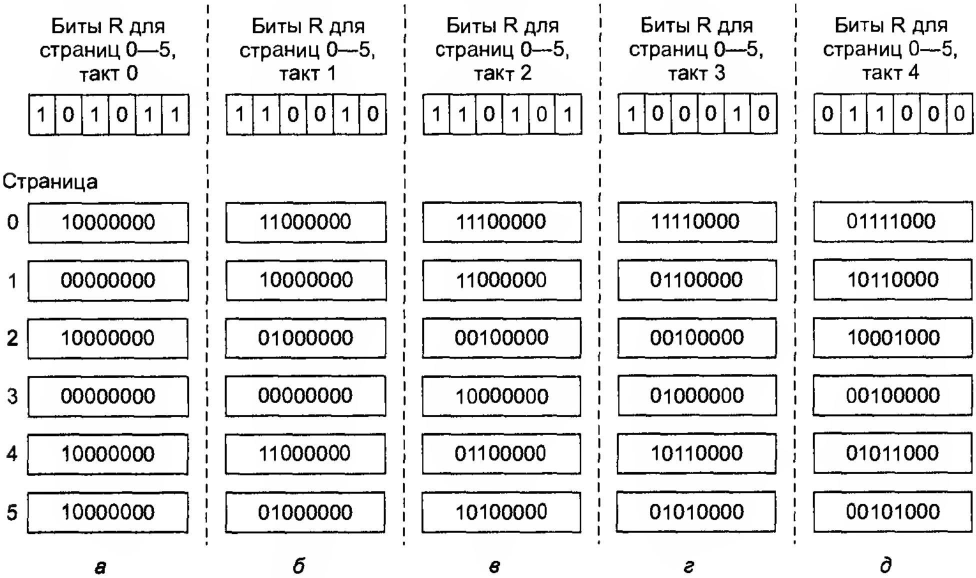


Рис. 4.18. Алгоритм старения программно моделирует алгоритм LRU. Здесь изображены шесть страниц после пяти тиков часов от (*а*) до (*д*)

Алгоритм «рабочий набор»

В простейшей схеме страничной подкачки в момент запуска процессов нужные им страницы отсутствуют в памяти. Как только центральный процессор пытается выбрать первую команду, он получает страничное прерывание, побуждающее операционную систему перенести в память страницу, содержащую первую инструкцию. Обычно следом быстро происходят страничные прерывания для глобальных переменных и стека. Через некоторое время в памяти скапливается большинство необходимых процессу страниц, и он приступает к работе с относительно небольшим количеством ошибок из-за отсутствия страниц. Этот метод называется **замещением страниц по запросу** (demand paging), потому что страницы загружаются в память по требованию, а не заранее.

Конечно, достаточно легко написать тестовую программу, систематически читающую все страницы в огромном адресном пространстве, вызывая так много страничных прерываний, что будет не хватать памяти для их обработки. К счастью, большинство процессов не работают таким образом. Они характеризуются **локальностью обращений**, означающей, что во время выполнения любой фразы процесс обращается только к сравнительно небольшой части своих страниц. Каждый проход многоходового компилятора, например, обращается только к части от общего количества страниц, и каждый раз к другой части.

Множество страниц, которое процесс использует в данный момент, называется **рабочим набором** ([89, 92]). Если рабочий набор целиком находится в памяти, процесс будет работать, не вызывая большого количества ошибок, до тех пор пока он не перейдет к другой фазе выполнения (то есть к следующему проходу компилятора). Если доступная память слишком мала для того, чтобы содержать полный рабочий набор, процесс вызовет много страничных прерываний и будет работать медленнее, так как выполнение инструкции занимает несколько наносекунд, а чтение страницы с диска обычно требует 10 мс. При скорости одна или две команды за 10 мс для завершения программы понадобятся века. Говорят, что программа, вызывающая страничное прерывание каждые несколько команд, **пробуксовывает** (thrashing) [90].

В многозадачных системах процессы часто перемещаются на диск (то есть все их страницы удаляются из памяти), чтобы позволить другим процессам получить доступ к центральному процессору. Возникает вопрос, что делать, когда процесс снова загружается в память. С формальной точки зрения делать ничего не нужно. Процесс будет вызывать одно за другим страничные прерывания до тех пор, пока не загрузится в память весь его рабочий набор. Проблема в том, что наличие 20, 100 или даже 1000 страничных прерываний при каждой загрузке процесса сильно замедляет работу системы и, кроме того, тратит впустую значительное количество времени работы центрального процессора, так как обработка страничного прерывания операционной системой требует нескольких миллисекунд работы процессора.

Поэтому многие системы со страничной организацией пытаются отслеживать рабочий набор каждого процесса и обеспечивают его нахождение в памяти до запуска процесса. Такой подход носит название **модели рабочего набора** [91]. Он разработан для того, чтобы значительно снизить процент страничных прерываний. Загрузка страниц *перед* тем, как разрешить процессу работать, также называется **опережающей подкачкой страниц** (prepaging). Заметьте, что рабочий набор изменяется с течением времени.

Давно известно, что большинство программ не обращаются к своему адресному пространству равномерно; чаще всего ссылки группируются на небольшом количестве страниц. Обращение к памяти может быть выборкой команды, данных или сохранением данных. В любой момент времени t существует множество страниц, использовавшихся за k последних обращений к памяти. Это множество $w(k, t)$ и есть рабочий набор. Так как все недавние обращения к памяти для $k > 1$ обязательно должны были обращаться ко всем страницам, использовавшимся для $k = 1$ обращения к памяти, то есть к последней и, возможно, еще к некоторым страницам, $w(k, t)$ является монотонно неубывающей функцией от k. Функция $w(k, t)$ ограничена для больших k, потому что программа не может обращаться к большему количеству страниц, чем содержится в ее адресном пространстве, кроме того, редкие программы обращаются ко всем страницам адресного пространства. На рис. 4.19 изображена зависимость размера рабочего набора от k.

Тот факт, что большинство программ случайным образом обращается к небольшому числу страниц, но это множество медленно изменяется во времени, объясняет быстрое возрастание функции в начале и затем медленное увеличение для больших k. Например, программа, которая выполняет цикл, затрагивающий две страницы и использующий данные на четырех страницах, может обращаться ко

всем шести страницам через каждые 1000 инструкций, но самое последнее обращение к некоторым другим страницам могло произойти миллионом команд раньше, во время начальной загрузки фазы. Вследствие этого асимптотического характера содержимое рабочего набора не чувствительно к выбранной величине *k*. Существует широкий диапазон значений *k*, для которых рабочий набор постоянен. Поскольку рабочий набор медленно меняется со временем, можно сделать разумное предположение, что те страницы, которые будут нужны для возобновления работы программы, составляют основу рабочего набора во время последней остановки процесса. Опережающая подкачка страниц заключается в загрузке рабочего набора до того, как процессу разрешается возобновиться.

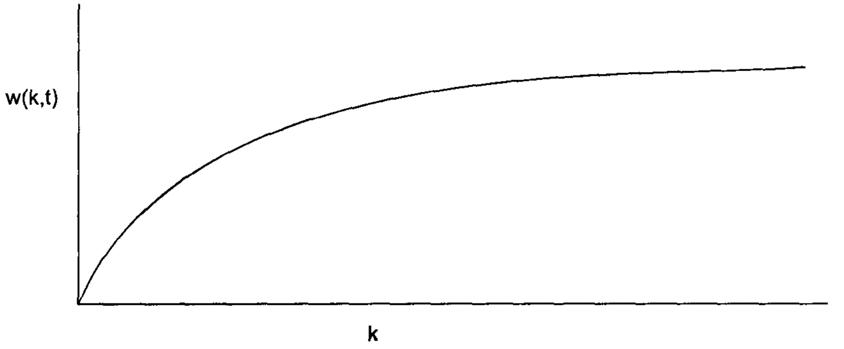


Рис. 4.19. Рабочий набор — это множество страниц, используемых k последними обращениями к памяти. Функция w(k, t) представляет собой размер рабочего набора в момент времени t

Чтобы реализовать модель рабочего набора, необходимо, чтобы операционная система отслеживала, какие страницы в нем находятся. Наличие этой информации также немедленно приводит к возможному алгоритму замещения страниц: когда происходит страничное прерывание, ищется и выгружается страница, не находящаяся в рабочем наборе. Для реализации такого алгоритма нужен точный метод определения того, какая страница находится в рабочем наборе, а какая в него не включена в любой заданный момент времени.

Как мы упоминали выше, рабочим набором является множество страниц, использовавшихся в *k* последних обращениях к памяти (некоторые авторы используют *k* последних страничных обращений, но выбор произволен). Для осуществления алгоритма «рабочий набор» заранее нужно назначить значение *k*. Как только некоторая величина выбрана, после каждого обращения к памяти набор страниц, используемых за предыдущие *k* обращений, определяется единственным образом.

Конечно, наличие действующего определения рабочего набора вовсе не означает, что существует эффективный способ отслеживания его в реальном времени, то есть во время выполнения программы. Можно было бы придумать сдвигающийся регистр длины *k*, который смещается влево на одну позицию при каждом обращении к памяти и записывает номер последней использовавшейся страницы в крайнюю правую позицию. Все *k* номеров страниц в сдвигающемся регистре входили бы в рабочий набор. Теоретически в момент страничного прерывания содержимое этого регистра могло бы считываться и сохраняться. Затем удалялись

бы дублирующиеся страницы. В результате мы бы получали рабочий набор. Однако поддержка сдвигающегося регистра и его обработка при страничном прерывании чрезвычайно дороги, поэтому данная техника никогда не употребляется на деле.

Вместо нее используются различные аппроксимации. Применяемый в большинстве случаев подход заключается в том, чтобы оставить идею подсчета *k* последних обращений к памяти и вместо этого использовать время выполнения программы. Например, вместо определения рабочего набора как множества страниц, на которые ссылались при предыдущих 10 млн обращений к памяти, мы можем определить его как множество страниц, использовавшихся в течение последних 100 мс времени выполнения. На практике это определение имеет тот же смысл, но намного упрощает реализацию алгоритма. Заметим, что для каждого процесса считается только его собственное время работы. Таким образом, если процесс стартовал во время *T* и занял процессор на 40 мс за реальное время *T* + 100 мс, для определения рабочего набора его время равно 40 мс. Время работы процессора, которое фактически использовал процесс с момента запуска, часто называется **текущим виртуальным временем**. При таком приближении рабочий набор процесса — это множество страниц, к которому он обращался за последние τ секунд виртуального времени.

Теперь рассмотрим алгоритм замещения страниц, основанный на рабочем наборе. Его базовая идея заключается в том, чтобы найти страницу, не включенную в рабочий набор, и выгрузить ее. На рис. 4.20 изображена часть таблицы страниц для некоторой машины. Поскольку в качестве кандидатов на удаление рассматриваются только те страницы, которые в настоящее время находятся в памяти, отсутствующие в памяти страницы этим алгоритмом игнорируются. Каждая запись содержит (по крайней мере) два элемента информации: приближенное время, в которое страница использовалась в последний раз, и бит *R* (обращения). Пустые белые прямоугольники символизируют другие поля, ненужные для данного алгоритма, такие как номер страничного блока, биты защиты и бит *M* (изменения).

Алгоритм работает следующим образом. Предполагается, что аппаратное обеспечение устанавливает биты *R* и *M*, как мы описывали выше. Предполагается также, что периодическое прерывание по таймеру вызывает запуск программы, очищающей бит *R* при каждом тике часов. При каждом страничном прерывании исследуется таблица страниц и ищется страница, подходящая для удаления из памяти.

В процессе обработки каждой записи проверяется бит *R*. Если он равен 1, текущее виртуальное время записывается в поле *Время последнего использования* (*Time of last use*) в таблице страниц, указывая, что страница использовалась в тот момент, когда произошло прерывание. Так как к странице было обращение в течение данного такта, ясно, что она находится в рабочем наборе и не является кандидатом на удаление (предполагается, что τ охватывает несколько тиков часов).

Если бит *R* равен 0, это означает, что к странице не было обращений в течение последнего тика часов и она может быть кандидатом на удаление. Чтобы понять, нужно ли ее выгружать, вычисляется ее возраст, то есть текущее виртуальное время минус ее *Время последнего использования*, и сравнивается с τ. Если возраст больше величины τ, это означает, что страница более не находится в рабочем наборе.

Она стирается, а на ее место загружается новая страница. Однако сканирование таблицы продолжается, обновляя остальные записи.

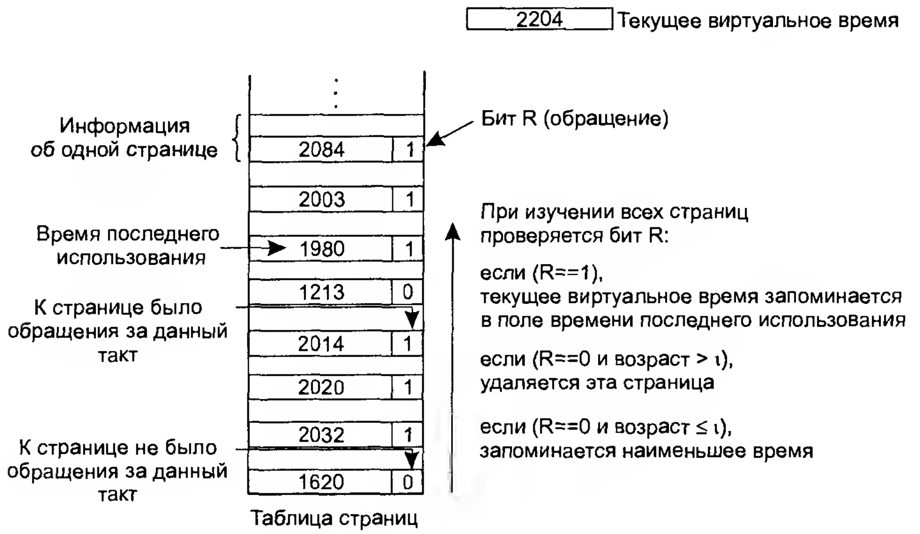


Рис. 4.20. Алгоритм «рабочий набор»

Если же бит R равен 0, но возраст страницы меньше или равен времени τ, это значит, что страница до сих пор находится в рабочем наборе. Она временно обходится, но страница с наибольшим возрастом запоминается (наименьшим значением *Времени последнего использования*). Если проверена вся таблица, а кандидат на удаление не найден, это означает, что все страницы входят в рабочий набор. В этом случае, если были найдены одна или больше страниц с битом $R = 0$, удаляется та из них, которая имеет наибольший возраст. В худшем случае ко всем страницам произошло обращение за время текущего такта часов (и, следовательно, все они имеют бит $R = 1$), тогда для удаления случайным образом выбирается одна из них, причем желательно чистая, если такая страница существует.

Алгоритм WSClock

Исходный алгоритм «рабочий набор» громоздок, так как при каждом страничном прерывании следует проверять таблицу страниц до тех пор, пока не определится местоположение подходящего кандидата. Усовершенствованный алгоритм, основанный на часовом алгоритме, но также использующий информацию рабочего набора, называется **WSClock** [54]. Благодаря простоте реализации и хорошей производительности этот алгоритм широко используется на практике.

Для него необходима структура данных в виде кольцевого списка страничных блоков, как в алгоритме «часы», что изображено на рис. 4.21, *а*. В исходном положении этот список пустой. Когда загружается первая страница, она добавляется в список. По мере прихода страниц они поступают в список, формируя кольцо.

Каждая запись, кроме бита *R* (показан) и бита *M* (не показан), содержит поле «*время последнего использования*» из базового алгоритма «рабочий набор».

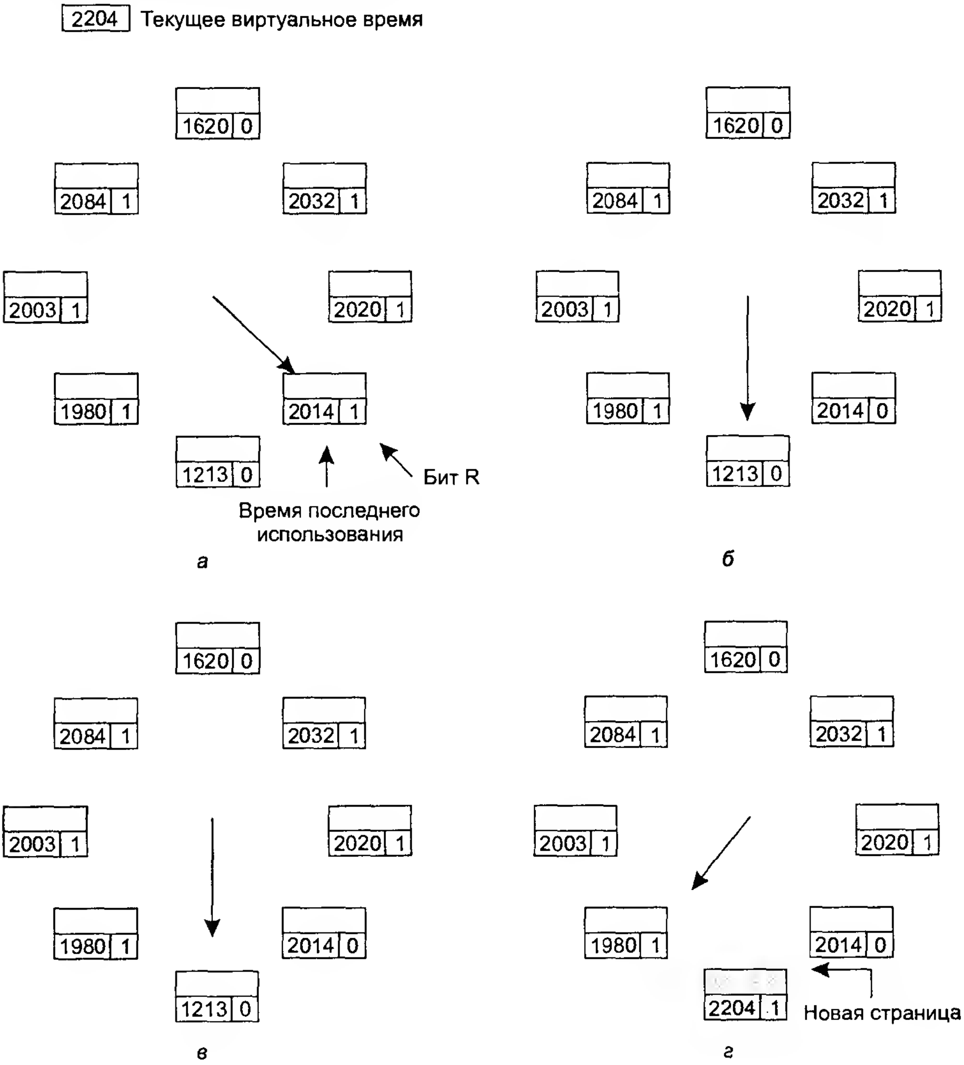


Рис. 4.21. Работа алгоритма WSClock: пример того, что происходит при бите *R* = 1 (*а*) и (*б*); пример для бита *R* = 0 (*в*) и (*г*)

Как и в случае алгоритма «часы», при каждом страничном прерывании первой проверяется та страница, на которую указывает стрелка. Если бит *R* равен 1, это значит, что страница использовалась в течение последнего такта часов, поэтому она не является идеальным кандидатом на удаление. Тогда бит *R* устанавливается на 0, стрелка передвигается на следующую страницу и для нее повторяется алгоритм. Состояние после такой последовательности действий продемонстрировано на рис. 4.21, *б*.

Теперь рассмотрим, что происходит, если страница, на которую указывает стрелка, имеет бит $R = 0$, как показано на рис. 4.21, *в*. Если возраст страницы больше величины τ и страница — чистая, то она не входит в рабочий набор и на диске есть ее действительная копия. Тогда в данный страничный блок просто загружается новая страница, как изображено на рис. 4.21, *г*. Если, напротив, страница «грязная», ее нельзя немедленно стереть, так как на диске нет ее последней копии. Чтобы избежать переключения процессов, запись на диск включается в график планирования, но стрелка сдвигается на позицию, и алгоритм продолжает работу со следующей страницей. Несмотря на то что «грязная» страница может быть старше, чистая находится ближе в ряду страниц, которые можно использовать немедленно.

Теоретически за один обход вокруг циферблата часов для всех страниц может оказаться запланированным ввод-вывод с диска. Чтобы уменьшить поток обмена с диском, можно установить предел, позволяющий быть записанными максимум *n* страницам. После достижения этой границы новые операции записи перестают включаться в график.

Что происходит, если стрелка обходит целый круг и возвращается к начальной точке? Существует два варианта:

1. Запланирована, по крайней мере, одна операция записи на диск.
2. Ни одной операции записи не запланировано.

В первом случае стрелка продолжает движение, отыскивая чистую страницу. Так как запланирована одна или больше операций записи на диск, со временем какая-нибудь из них будет выполнена, и соответствующая страница будет помечена как чистая. Выгружается первая попавшаяся чистая страница. Это не обязательно та страница, запись которой запланирована первой, потому что драйвер диска может изменить порядок работы с диском, чтобы оптимизировать его производительность.

Во втором случае все страницы находятся в рабочем наборе, иначе планировалась бы, по крайней мере, одна операция записи. За недостатком дополнительной информации проще всего предъявить права на любую чистую страницу и использовать ее. Расположение чистой страницы могло бы отслеживаться во время «чистки». Если в памяти нет чистых страниц, тогда выбирается текущая страница и переписывается на диск.

Алгоритмы замещения страниц, резюме

Мы рассмотрели множество различных алгоритмов замещения страниц. В этом разделе мы кратко подведем итоги вышесказанного. Список обсужденных алгоритмов представлен в табл. 4.2.

Оптимальный алгоритм заменяет ту страницу, обращение к которой производилось раньше других, находящихся в данный момент в памяти. К сожалению, не существует способа определения того, какая страница будет последней, поэтому данный алгоритм не может использоваться на практике. Но он полезен в качестве тестовой задачи, относительно которой можно оценивать другие алгоритмы.

Алгоритм NRU (не использовавшаяся в последнее время страница) делит страницы на четыре класса в зависимости от состояния битов *R* и *M*. Выбирается любая

страница из класса с наименьшим номером. Этот алгоритм легко реализуется, но он является очень грубым. Существуют лучшие схемы.

Таблица 4.2. Алгоритмы замещения страниц, описанные в тексте

Алгоритм	Комментарии
Оптимальный	Не осуществим, но полезен в качестве тестовой задачи
NRU (не использовавшаяся в последнее время страница)	Очень грубый
FIFO (первым прибыл, первым обслужен)	Может выгрузить важные страницы
Вторая попытка	Значительное усовершенствование FIFO
Часы	Реалистичный
LRU (страница, не использовавшаяся дольше всего)	Отличный алгоритм, но его сложно осуществить целиком
NFU (редко использовавшаяся страница)	Довольно грубое приближение алгоритма LRU
Старение	Эффективный алгоритм, хорошо аппроксимирующий алгоритм LRU
Рабочий набор	Немного дорог для реализации
WSClock	Хороший рациональный алгоритм

Алгоритм FIFO (первым прибыл – первым обслужен) отслеживает порядок загрузки страниц в память, храня их в связном списке. При этом удаление старейшей страницы становится тривиальным, но эта страница может использоваться в данный момент, поэтому алгоритм FIFO представляет собой плохой выбор.

Алгоритм «вторая попытка» – это модификация алгоритма FIFO, он перед удалением страницы из памяти проверяет, используется ли она в данный момент. Если да, то страница пропускается. Такое изменение сильно повышает производительность. Алгоритм «часы» представляет собой всего лишь другое осуществление алгоритма «второй попытки». Он имеет те же самые характеристики производительности, но требует немного меньше времени на выполнение алгоритма.

Алгоритм LRU (страница, не использовавшаяся дольше всего) – это отличный алгоритм, но его нельзя осуществить без специального аппаратного обеспечения. Если подобное оборудование недоступно, алгоритм невозможно использовать. Алгоритм NFU (редко использовавшаяся страница) представляет собой грубую попытку аппроксимации алгоритма LRU. Он не очень хорош. Но существует алгоритм «старения», который намного лучше аппроксимирует алгоритм LRU и может быть эффективно реализован. Это замечательный выбор.

Последние два алгоритма используют рабочий набор. Алгоритм «рабочий набор» обладает приемлемой производительностью, но дорог в реализации. Алгоритм WSClock – это вариант, который не только дает достойную производительность, но его также достаточно просто реализовать.

В итоге двумя наилучшими алгоритмами являются «старение» и WSClock. Они основаны на алгоритме LRU и понятии рабочего набора соответственно. Оба обеспечивают хорошую постраничную подкачку и могут быть реализованы за разум-

ную цену. Существует еще несколько алгоритмов, но для практических целей эти два являются, вероятно, наиболее важными.

Моделирование алгоритмов замещения страниц

За годы было проведено несколько работ, посвященных теоретическому моделированию алгоритмов замещения страниц. В данном разделе мы обсудим некоторые из этих идей, чтобы увидеть, как работает процесс моделирования.

Аномалия Билэди

Интуитивно может показаться, что чем больше страничных блоков имеет память, тем меньше будет происходить страничных прерываний. Достаточно удивителен тот факт, что это не всегда так. Билэди (Belady) и другие исследователи в своей работе [23] описали обнаруженный ими контрпример, в котором алгоритм FIFO вызывал больше страничных прерываний при четырех страничных блоках, чем при трех. Эта странная ситуация стала известна как **аномалия Билэди**. Она проиллюстрирована на рис. 4.22 для программы с пятью виртуальными страницами, пронумерованными от 0 до 4. Буквы «Р» показывают, какие обращения вызывают страничные прерывания. Обращения к страницам происходят в следующем порядке:

0 1 2 3 0 1 4 0 1 2 3 4

На рис. 4.22, *а* показано, как при наличии трех страничных блоков вызывается в целом девять страничных прерываний. На рис. 4.22, *б* изображены десять страничных прерываний при работе с четырьмя страничными блоками.

Все страничные блоки изначально пусты

		0	1	2	3	0	1	4	0	1	2	3	4	
Самая младшая страница		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Самая старшая страница				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 страничных прерываний

а

		0	1	2	3	0	1	4	0	1	2	3	4	
Самая младшая страница		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Самая старшая страница				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 страничных прерываний

б

Рис. 4.22. Аномалия Билэди: алгоритм FIFO при работе с тремя страничными блоками (*а*); алгоритм FIFO при наличии четырех страничных блоков (*б*)

Магазинные алгоритмы

Аномалия Билэди настолько потрясла многих исследователей в области кибернетики, что они начали изучать данную ситуацию, и это привело к развитию целой теории алгоритмов подкачки страниц и их свойств. Хотя большая часть данных исследований лежит далеко за пределами нашей книги, ниже мы кратко рассмотрим основные моменты. За более подробной информацией следует обратиться к [220].

Вся работа началась с наблюдения, что каждый процесс с момента запуска формирует последовательность обращений к памяти. Любая ссылка к памяти соответствует определенной виртуальной странице. Таким образом, концептуально доступ процесса к памяти можно описать (упорядоченным) списком номеров страниц. Этот список называется последовательностью или **строкой обращений** (reference string) и играет главную роль во всей теории. Для простоты далее мы будем рассматривать вариант машины с одним процессом, то есть когда каждая машина имеет единственную определенную последовательность обращений (при нескольких процессах мы должны были бы принять во внимание чередование их строк обращений вследствие многозадачности).

Систему со страничной организацией памяти можно охарактеризовать следующими тремя объектами:

1. Последовательность обращений для выполняемого процесса.
2. Алгоритм замещения страниц.
3. Количество доступных в памяти страничных блоков m.

Мысленно мы можем представить себе абстрактный интерпретатор, работающий следующим образом. Он поддерживает внутренний массив M, отслеживающий состояние памяти. Количество элементов массива равно количеству виртуальных страниц процесса, это число мы назовем n. Массив M разделен на две части. В верхней части, куда входит m записей, расположены все страницы, которые в данный момент находятся в памяти. В нижней части размером $n - m$ записей содержатся номера всех страниц, к которым когда-то произошло обращение, но они были выгружены из памяти и в данный момент в ней отсутствуют. В исходном положении массив M пуст, так как процесс еще не обращался ни к одной странице, и в памяти страниц тоже еще нет.

После запуска процесс начинает вызывать страницы из строки обращений по одной. Как только появляется очередная страница, интерпретатор проверяет, находится ли страница в памяти (то есть в верхней части массива M). Если нет, происходит страничное прерывание. Если в памяти есть пустой сегмент (то есть верхняя часть массива M содержит меньше, чем m записей), страница загружается и добавляется в верхнюю часть массива M. Подобная ситуация возникает только на начальной стадии выполнения. Если память заполнена (то есть верхняя часть массива M содержит m записей), то, чтобы удалить страницу из памяти, активизируется алгоритм замещения страниц. В модели все происходит так: одна страница перемещается из верхней части массива M в его нижнюю часть, а требуемая страница входит наверх. Кроме того, верхняя и нижняя части массива могут быть упорядочены отдельно друг от друга.

Чтобы прояснить функционирование интерпретатора, рассмотрим конкретный пример, использующий алгоритм замещения страниц LRU. Виртуальное адресное пространство имеет восемь страниц, а в физической памяти есть четыре страничных блока. Сверху на рис. 4.23 изображена последовательность обращений, состоящая из 24 страниц:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

Ниже последовательности обращений расположена таблица из 25 столбцов и 8 строк. Первый столбец не заполнен, так как он отражает состояние массива *M* до запуска процесса. Каждый следующий столбец демонстрирует массив *M* после того, как одна из страниц была извлечена обращением к ней и обработана алгоритмом подкачки страниц. Жирный контур отделяет верхнюю часть массива *M*, то есть первые четыре элемента, соответствующие страничным блокам в памяти. Страницы внутри жирной рамки находятся в памяти, страницы, расположенные ниже, были выгружены на диск.

Последовательность обращений		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
			0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
				0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
					0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
						0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
							0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
								0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Страничное прерывание		P	P	P	P	P	P	P		P					P			P						P	
Последовательность расстояний		∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Рис. 4.23. Состояние массива памяти M после обработки каждого элемента строки обращений. Последовательность расстояний будет обсуждаться в следующем разделе

Первой в последовательности обращений является страница 0, поэтому она помещается на самый верх памяти, как показано во втором столбце. Следующая страница под номером 2 попадает в верхнюю ячейку третьего столбца. Это действие сдвигает вниз страницу 0. В данном примере заново загружаемая страница всегда занимает верхнюю ячейку, а все остальное по необходимости сдвигается вниз.

Каждая из первых семи страниц в последовательности обращений вызывает страничное прерывание. Первые четыре можно обработать, не удаляя страницы из памяти, но начиная со страницы 5 загрузка новой страницы требует удаления старой.

Второе обращение к странице 3 не вызывает страничного прерывания, потому что она уже находится в памяти. Тем не менее интерпретатор убирает ее с того места, где она располагалась, и помещает в верхнюю ячейку столбца, как показано на рисунке. Процесс продолжает работу некоторое время, до тех пор, пока не происходит обращение к странице 5. Эта страница переносится из нижней части массива *M* в верхнюю (то есть она загружается в память с диска). Всякий раз, когда

страница, к которой обращается процесс, не находится внутри рамки, очерченной жирной линией, происходит страничное прерывание, что отмечено буквами «П» в строке под таблицей.

Теперь кратко перечислим некоторые свойства этой модели. Во-первых, когда происходит обращение к странице, она всегда перемещается в верхнюю запись массива M. Во-вторых, если запрашиваемая страница уже находилась в массиве M, все страницы выше нее сдвигаются на одну позицию вниз. Переход из рамки за ее пределы соответствует удалению страниц из памяти. В-третьих, страницы, находящиеся ниже объекта обращения, не перемещаются. Таким образом, содержимое массива M в точности представляет собой компоненты алгоритма LRU.

Хотя в этом примере используется алгоритм LRU, данная модель с тем же успехом работает и с другими схемами. В частности, существует один класс алгоритмов, которые представляют особенный интерес, они обладают свойством:

$$M(m, r) \subseteq M(m + 1, r),$$

где число m обозначает количество страничных блоков, а r — это индекс в последовательности обращений. Изображенное выше выражение означает, что множество страниц, после r обращений попавших в верхнюю часть массива M, для памяти, имеющей m страничных блоков, также входит в массив M, если память состоит из $m + 1$ страничных блоков. Другими словами, если мы увеличим размер памяти на один страничный блок и выполним процесс заново, то в каждый момент времени все страницы, присутствовавшие в памяти во время первой обработки, при втором запуске будут также находиться в памяти вместе с еще одной дополнительной страницей.

Если изучить пример на рис. 4.23 и немного подумать о том, как он функционирует, должно стать ясно, что алгоритм LRU удовлетворяет данному условию. Некоторые другие алгоритмы (например, оптимальный алгоритм замещения) также обладают этим свойством, но алгоритм FIFO его не имеет. Алгоритмы, удовлетворяющие условию, наложенному на массив M, называются **магазинными алгоритмами** (stack algorithms). Они не подвержены аномалии Билэди и, соответственно, намного более любимы теоретиками, занимающимися виртуальной памятью.

Строка расстояний

Часто для магазинных алгоритмов удобно представить последовательность обращений в более абстрактном виде, чем фактические номера страниц. С этого момента обращение к странице будет обозначаться с помощью расстояния от верха стека, где расположена запрашиваемая страница. Например, обращение к странице 1 в последнем столбце на рис. 4.23 равносильно ссылке на страницу, имеющую расстояние 3 от вершины стека (потому что страница 1 *перед* запросом находилась на третьем месте). О страницах, к которым еще не было обращений и поэтому они еще не попали в стек памяти (то есть они еще не находятся в массиве M), говорят, что они имеют расстояние ∞ (бесконечность). Строка расстояний для примера на рис. 4.23 изображена внизу рисунка.

Заметим, что строка расстояний зависит не только от последовательности обращений, но и от алгоритма подкачки страниц. При одной и той же последовательности обращений различные алгоритмы замещения страниц могут выбирать для удаления разные страницы. В результате возникает свой порядок стека для каждого алгоритма.

Статистические свойства последовательности расстояний сильно влияют на производительность алгоритма. На рис. 4.24, *а* представлена функция, обозначающая плотность вероятности для вхождений страниц в (воображаемую) строку расстояний *d*. Большинство попаданий страниц находится между 1 и *k*. Если в памяти всего *k* страничных блоков, то страничные прерывания происходят редко.

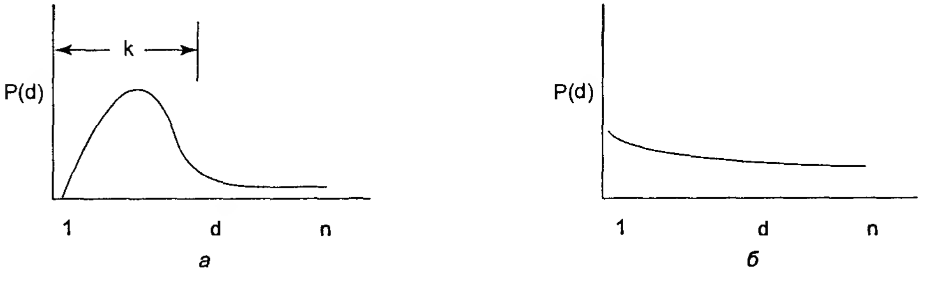


Рис. 4.24. Плотность вероятности для двух гипотетических строк расстояний

Напротив, на рис. 4.24, *б* обращения к памяти так разбросаны, что единственный способ избежать огромного количества страничных прерываний — это предоставить программе столько страничных блоков, сколько она использует виртуальных страниц. Если вам приходится работать с подобными программами, значит, у вас, видимо, просто плохая карма.

Прогнозирование частоты страничных прерываний

Одно из приятных свойств последовательности расстояний заключается в том, что ее можно использовать для прогнозирования количества страничных прерываний, которые могут произойти в памяти различного размера. Мы продемонстрируем, как можно выполнить подобные вычисления на основе примера с рис. 4.23. Нашей целью является следующее: сделать один проход по строке расстояний и по собранной информации суметь предсказать, сколько страничных прерываний мог бы вызвать процесс в памяти размером в 1, 2, 3, ..., *n* страничных блоков, где *n* — это количество виртуальных страниц в адресном пространстве процесса.

Алгоритм начинается с изучения последовательности расстояний, страница за страницей. Он подсчитывает, сколько раз встречается число 1, число 2 и т. д. Пусть число *i* встречается в строке расстояний C_i количество раз. Вектор *C* для последовательности расстояний на рис. 4.23 изображен на рис. 4.25, *а*. В этом примере получилось так, что четыре раза происходит обращение к странице, уже находящейся на вершине стека. Три раза запрашивается следующая страница и т. д. Пусть C_∞ — это количество раз, которое встречается символ ∞ в последовательности расстояний.

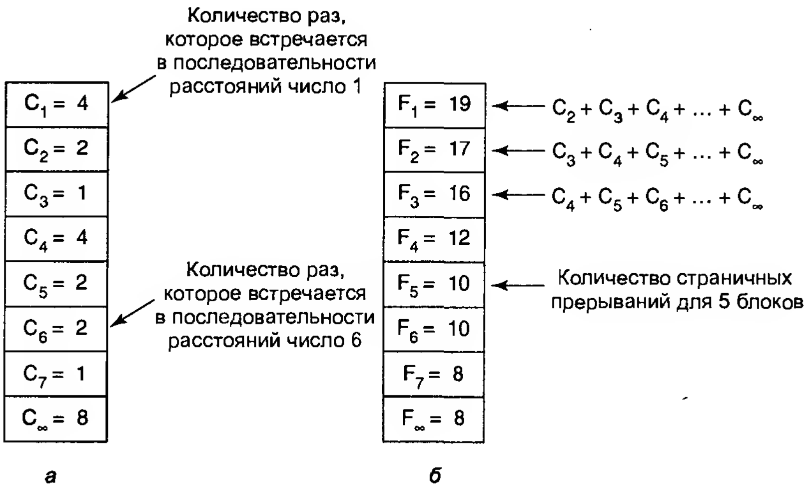


Рис. 4.25. Вычисление количества страничных прерываний из последовательности расстояний: вектор *C* (*а*); вектор *F* (*б*)

Теперь вычислим вектор F в соответствии с формулой

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty.$$

Величина F_m обозначает количество страничных прерываний, которое произойдет для заданной последовательности расстояний и m страничных блоков. На рис. 4.25, *б* показан вектор F для строки расстояний, представленной на рис. 4.23. Например, величина F_1, равная 20, означает, что если память состоит всего лишь из одного страничного блока, то из 24-х обращений в последовательности вызовут страничное прерывание все, кроме четырех, которые запрашивают ту же страницу, что и предыдущая ссылка.

Чтобы увидеть, как работает формула, вернемся к рамке, очерченной жирной линией на рис. 4.23. Пусть m — это количество страниц в верхней части массива M. Страничное прерывание происходит всякий раз, когда элемент последовательности расстояний равен $m+1$ или больше. В написанной выше формуле суммируется то количество раз, которое встречаются в последовательности такие элементы. Эта модель также может использоваться и для других прогнозов [220].

Вопросы разработки систем со страничной организацией памяти

В предыдущих разделах мы объяснили, как работает подкачка по страницам, представили несколько основных алгоритмов замещения страниц и показали, как их моделировать. Но знания голой механики недостаточно. Чтобы разработать хорошо работающую систему, вы должны знать о ней намного больше. Разница такая же, как между человеком, который знает о том, как ходят ладья, конь, слон и другие

шахматные фигуры, и хорошим шахматистом. В следующих разделах мы рассмотрим другие вопросы, которые должны принимать во внимание разработчики операционных систем для того, чтобы получить достойную производительность системы со страничной организацией памяти.

Политика распределения памяти: локальная и глобальная

В предыдущих разделах мы обсудили несколько алгоритмов, ищущих страницу для замещения, когда происходит прерывание. Основной вопрос, связанный с этим выбором (который мы тщательно обходили до сих пор): как должна быть распределена память между параллельными конкурирующими работоспособными процессами?

Обратим внимание на рис. 4.26, *а*. Здесь три процесса, *A*, *B* и *C*, составляют набор работоспособных процессов. Предположим, процесс *A* вызвал страничное прерывание. Должен ли алгоритм замещения страниц пытаться найти наиболее давно использовавшуюся страницу, учитывая только шесть страниц, предоставленные в данный момент процессу *A*, или же он должен рассматривать все страницы памяти? Если алгоритм производит поиск только среди страниц процесса *A*, наименьший возраст имеет страница *A*5, и мы получаем ситуацию, изображенную на рис. 4.26, *б*.

С другой стороны, если удаляется страница с наименьшим возрастом, независимо от того, к какому процессу она относится, то будет выбрана страница *B*3, и система попадет в состояние, показанное на рис. 4.26, *в*. Алгоритм на рис. 4.26, *б* называется **локальным**, а про схему на рис. 4.26, *в* говорят, что это **глобальный** алгоритм замещения страниц. Локальные алгоритмы соответствуют размещению каждого процесса в фиксированной области памяти. Глобальные алгоритмы динамически распределяют страничные блоки между выполняющимися процессами. Таким образом, количество страничных блоков, предоставленных каждому процессу, изменяется со временем.

В целом глобальные алгоритмы работают лучше, особенно если размер рабочего набора может изменяться за время жизни процесса. Если используется локальный алгоритм и рабочий набор увеличивается в размере, в результате мы получим пробуксовку, даже когда в системе существует достаточное количество свободных страничных блоков. Если рабочий набор уменьшается, при локальном алгоритме часть памяти потратится впустую. Если же используется глобальный алгоритм, система должна непрерывно выносить решения о том, сколько страничных блоков нужно предоставить каждому процессу. Можно наблюдать за размером рабочего набора с помощью битов возраста страниц, но этот метод не всегда позволяет избежать пробуксовки. Рабочий набор может изменяться в размере за микросекунды, тогда как возрастные биты являются грубым усреднением за тик часов.

Другой способ состоит в том, чтобы иметь в системе алгоритм для распределения страничных блоков между процессами. Например, можно периодически определять количество работающих процессов и предоставлять каждому равную часть памяти. Соответственно, при наличии доступных (то есть не принадлежа-

щих операционной системе) 12 416 страничных блоках и 10 процессах каждый процесс получит 1241 блок. Оставшиеся 6 блоков поступают в резерв и могут использоваться в тот момент, когда происходит страничное прерывание.

а	Возраст	б	в
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

Рис. 4.26. Локальный алгоритм замещения страниц против глобального: исходная конфигурация (*а*); локальное замещение страниц (*б*); глобальное замещение страниц (*в*)

Хотя этот метод кажется справедливым, существует небольшой шанс, что процессы размером 10 Кбайт и 300 Кбайт получат равные области памяти. Вместо этого можно предоставлять страницы пропорционально абсолютному размеру каждого процесса, тогда процесс размером 300 Кбайт получит долю памяти в 30 раз больше, чем процесс размером 10 Кбайт. При этом разумно отдавать каждому процессу некоторый минимум, чтобы он мог работать независимо от своего размера. На некоторых машинах, например, одиночная команда процессора, состоящая из двух операндов, может нуждаться в целых шести страницах, потому что сама команда, операнд-источник и операнд-приемник могут пересекать границы страниц. Если предоставить только пять страниц, программа, содержащая подобную инструкцию, вообще не сможет выполняться.

Если используется глобальный алгоритм, допустимо запускать каждый процесс с некоторым количеством страниц, пропорциональным его размеру, но распределение памяти можно динамически изменять во время работы. Алгоритм **PFF** (Page Fault Frequency — частота страничных прерываний) предоставляет один из способов управления размещением процессов в памяти. Он говорит, когда увеличивать или уменьшать количество страниц, предоставленных процессу, но не упоминает о том, какую страницу замещать по прерыванию. Этот алгоритм только контролирует размер набора страниц, назначенных процессу.

Для большого класса схем замещения страниц, включая алгоритм LRU, известно, что частота прерываний уменьшается при увеличении числа предоставленных страниц, как мы обсуждали выше. Эта посылка лежит в основе алгоритма PFF. Данное свойство иллюстрирует рис. 4.27.

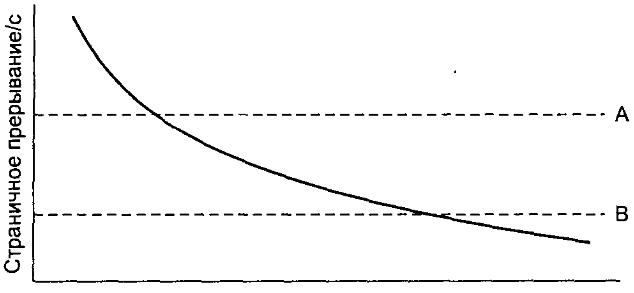


Рис. 4.27. Частота страничных прерываний как функция от количества предоставленных процессу страничных блоков

Частота страничных прерываний измеряется напрямую: просто считается количество прерываний в секунду, возможно также с вычислением скользящего среднего за несколько последних секунд. Существуют достаточно легкие способы такого подсчета, например, к текущему среднему значению прибавляется значение в секундах в данный момент и делится на два. Пунктирная линия, обозначенная буквой *A*, соответствует частоте страничных прерываний, выше которой она недопустимо высока, поэтому увеличивается количество страничных блоков, предоставленных прерванному процессу, с целью уменьшения процента прерываний. Пунктирная линия *B* соответствует очень низкой частоте страничных прерываний, позволяющей сделать вывод, что процесс занимает слишком много памяти. В этом случае у него можно забрать несколько страничных блоков. Таким образом, алгоритм PFF пытается сохранить частоту подкачки страниц для каждого процесса внутри допустимых границ.

Важно отметить, что некоторые алгоритмы замещения страниц могут работать как с локальной политикой замещения страниц, так и с глобальной. Например, алгоритм FIFO может выгрузить самую старую страницу во всей памяти (глобальный алгоритм) или старейшую страницу, принадлежащую данному процессу (локальный алгоритм). Аналогично, алгоритм LRU или некоторые его аппроксимации могут заменить страницу, не использовавшуюся дольше всего, выбираемую из всей памяти (глобальный алгоритм) или выбираемую среди страниц, соответствующих выполняемому в текущий момент процессу (локальный алгоритм). Выбор между локальной и глобальной политикой в некоторых случаях не зависит от алгоритма.

С другой стороны, для некоторых алгоритмов замещения страниц имеет смысл только локальная стратегия. В частности, алгоритмы «рабочий набор» и WSClock относятся к конкретному процессу и должны применяться именно в этом контексте. Реально для машины в целом не существует понятия рабочего набора, и если попытаться использовать объединение всех рабочих наборов, то это непременно приведет к потере характерных свойств и хорошо работать не будет.

Регулирование загрузки

Даже при работе с лучшим алгоритмом замещения страниц и оптимальным глобальным распределением страничных блоков между процессами может произойти так,

что система начнет буксовать. В самом деле, когда сумма рабочих наборов всех процессов превышает размеры памяти, можно ожидать пробуксовки. Симптомом этой ситуации является показание алгоритма PFF, что некоторые процессы нуждаются в дополнительной памяти, но в системе нет процессов, требующих меньше памяти. В таком случае не существует способа предоставить больше памяти тем процессам, которым это необходимо, без повреждения каких-то других процессов. Есть только одно реальное решение: временно избавиться от некоторых процессов.

Уменьшить количество конкурирующих за использование памяти процессов можно, выгрузив некоторые из них целиком на диск и освободив все занимаемые ими страницы. Например, один процесс можно полностью переместить на диск, а его страничные блоки разделить между буксующими процессами. Если пробуксовка прекращается, то система может работать некоторое время в таком состоянии. Если не прекращается, то выгружается следующий процесс и т. д. до тех пор, пока не закончится пробуксовка. Таким образом, даже при страничной организации памяти и страничной подкачке все еще необходима обычная подкачка (свопинг), только теперь она используется для того, чтобы уменьшить потенциальную потребность в памяти, а не с целью возврата блоков в систему для непосредственного использования.

Обычная подкачка процессов для того, чтобы ослабить загрузку памяти, напоминает двухуровневое планирование, при котором часть процессов помещается на диск и используется временный планировщик для составления графика работы оставшихся процессов. Понятно, что две идеи можно комбинировать, то есть выгрузить достаточное количество процессов на диск, чтобы сделать приемлемой частоту страничных прерываний. Периодически некоторые процессы подкачиваются с диска, и тогда туда целиком выгружаются другие процессы.

Еще одним фактором, который следует принять во внимание, является степень многозадачности. Как мы видели на рис. 4.4, когда количество процессов в основной памяти слишком мало, центральный процессор может простаивать значительные периоды времени. Когда нужно принять решение о том, какой процесс выгружать из памяти, это наблюдение является аргументом в пользу учета не только размера процесса и частоты подкачки страниц. Оно важно и для определения того, является ли он процессом, ограниченным возможностями процессора, или процессом, ограниченным скоростью ввода-вывода, а также аналогичных свойств остальных процессов.

Размер страницы

Зачастую размер страницы является параметром, выбираемым операционной системой. Даже если аппаратное обеспечение предусматривает, например, размер страницы 512 байт, операционная система может просто рассматривать страницы 0 и 1, 2 и 3, 4 и 5 и т. д. как страницы размером 1 Кбайт, всегда предоставляя для них два последовательных страничных блока.

Определение наилучшего размера страниц требует уравновешивания нескольких параллельных факторов. Поэтому не существует абсолютного оптимального решения. Прежде всего, возникают два довода в пользу маленького размера страниц. Случайно выбранный текст, данные или сегмент стека не заполняют целое количество страниц. В среднем половина последней страницы оказывается пустой,

и это дополнительное пространство пропадает. Такие потери называют **внутренней фрагментацией**. Если в памяти n сегментов, а размер страницы равен p байтам, $np/2$ байт будет потрачено впустую в результате внутренней фрагментации. Это разумный аргумент в пользу страниц небольшого размера.

Другой довод становится очевидным, если мы представим себе программу, состоящую из восьми последовательных этапов, по 4 Кбайт каждый. При размере страницы 32 Кбайт программе должно быть постоянно выделено 32 Кбайт. При размере страницы 16 Кбайт ей необходимо только 16 Кбайт. При размере страницы 4 Кбайт или меньше программа требует всего лишь 4 Кбайт к любой момент времени. То есть большой размер страницы скорее, чем маленький, станет причиной того, что в памяти находится неиспользуемая часть страницы.

С другой стороны, небольшой размер страницы означает, что программам будет нужно много страниц, следовательно, огромная таблица страниц. Программа размером 32 Кбайт требует всего четыре страницы по 8 Кбайт и 64 страницы по 512 байт. Как правило, страница за раз переносится на диск и с него, при этом большая часть времени уходит на поиск цилиндра и задержку вращения, так что перемещение маленькой страницы занимает почти столько же времени, сколько и большой. Может потребоваться 64 × 10 мс, чтобы загрузить 64 страницы размером 512 байт, и всего лишь 4 × 12 мс для загрузки четырех страниц по 8 Кбайт.

На некоторых машинах таблица страниц должна записываться в аппаратные регистры каждый раз, когда процессор переключается от одного процесса к другому. Если на таком компьютере страница имеет маленький размер, то время, требующееся для загрузки таблицы, будет увеличиваться пропорционально уменьшению размера страницы. Более того, пространство, занятое таблицей страниц, также возрастает с уменьшением страницы.

Этот последний момент можно проанализировать математически. Пусть средний размер процесса равен s байт, а страницы — p байт. Кроме того, предположим, что запись для каждой страницы требует e байт. Тогда приблизительное количество страниц, необходимое для процесса, равно s/p, что займет se/p байт для таблицы страниц. Потеря памяти в последней странице процесса вследствие внутренней фрагментации равна $p/2$. Таким образом, общие накладные расходы вследствие поддержки таблицы страниц и потери от внутренней фрагментации равны сумме этих двух составляющих:

$$\text{расход} = se/p + p/2.$$

Первое слагаемое (размер таблицы страниц) увеличивается при уменьшении размера страницы. Второе слагаемое (внутренняя фрагментация) при увеличении размера страницы возрастает. Оптимальный вариант должен находиться где-то посередине. Если взять первую производную по переменной p и приравнять ее к нулю, мы получим равенство:

$$-se/p^2 + 1/2 = 0.$$

Из этого равенства мы можем получить формулу, дающую оптимальный размер страниц (принимая во внимание только потери памяти на фрагментацию и размер таблицы страниц). В результате получится:

$$p = \sqrt{2se}.$$

Для среднего размера процесса $s = 1$ Мбайт и размера записи в таблице страниц $e = 8$ байт оптимальный размер страницы будет равен 4 Кбайт. В серийно выпускаемых компьютерах использовался размер страниц в диапазоне от 512 байт до 64 Кбайт. Раньше обычно употреблялась величина 1 Кбайт, но в наши дни более часто встречаются 4 Кбайт или 8 Кбайт. Так как памяти становится больше, то размер страниц также имеет тенденцию роста (но зависимость не линейная). Увеличение вчетверо размера оперативной памяти редко удваивает размер страницы.

Отдельные пространства команд и данных

Большинство компьютеров имеют единое адресное пространство, в котором содержатся и программы, и данные к ним, как показано на рис. 4.28, *а*. Если адресное пространство достаточно вместительно, все прекрасно работает. Но часто адресное пространство имеет слишком маленький размер, что вынуждает программистов ломать головы над тем, как разместить в нем все необходимое.

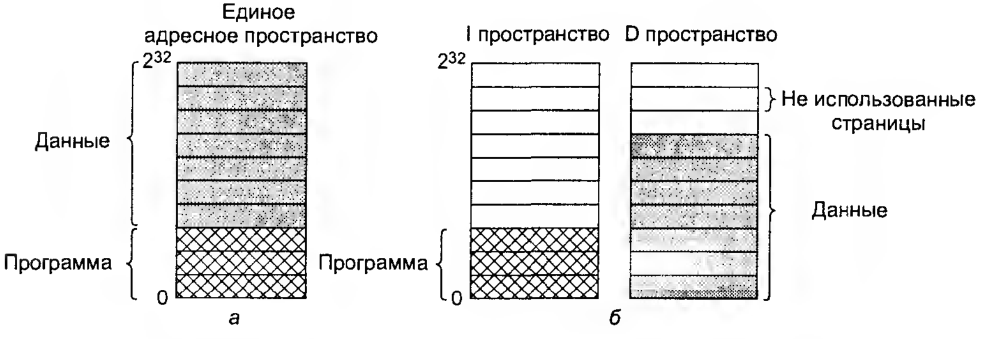


Рис. 4.28. Одно адресное пространство (*а*); отдельные I и D пространства (*б*)

Одно из решений проблемы, впервые примененное на (16-разрядном) компьютере PDP-11, заключается в разделении адресных пространств для инструкций (или команд, то есть текста программы) и данных. Они называются **I-пространство** (instruction — инструкция) и **D-пространство** (data — данные). Каждое адресное пространство расположено в диапазоне от 0 и до некоторого максимума, обычно до $2^{16}-1$ или $2^{32}-1$. На рис. 4.28, *б* изображены оба пространства. Компоновщик должен знать, когда используются отдельные I- и D-пространства, потому что если они существуют, адреса данных настраиваются на виртуальный адрес 0 вместо того, чтобы начинаться после программы.

В компьютере, устроенном таким образом, оба адресных пространства могут иметь страничную организацию независимо друг от друга. Каждое из них обладает своей собственной таблицей страниц и собственным отображением виртуальных страниц на физические страничные блоки. Когда аппаратура хочет выбрать команду, она знает, что должна использовать I-пространство и таблицу страниц I-пространства. Аналогично, обращения к данным должно происходить через таблицу страниц D-пространства. Кроме этого различия, поддержка отдельных I- и D-пространств не вносит каких-либо специальных осложнений и удваивает доступное адресное пространство.

Совместно используемые страницы

Еще один вопрос разработки — это совместный доступ или разделение страниц. В больших многозадачных системах часто случается, что в одно и то же время несколько пользователей работают с одной программой. Ясно, что более эффективно использовать страницы совместно, чтобы избежать одновременного присутствия в памяти двух копий одной и той же страницы. К сожалению, не все страницы разделяемы. В частности, страницы только для чтения, такие как текст программы, можно использовать совместно, а страницы с данными — нельзя.

Если поддерживаются отдельные I- и D-пространства, относительно просто обеспечить общий доступ к программам, разрешив двум или более процессам использовать одну и ту же таблицу страниц для их I-пространства и различные таблицы страниц для их D-пространств. Обычно при такой реализации поддержки совместного доступа таблицы страниц и структуры данных не зависят от таблицы процесса. Тогда каждый процесс в своей таблице процесса имеет два указателя: один на таблицу страниц I-пространства и другой на таблицу страниц D-пространства, как показано на рис. 4.29. Когда планировщик выбирает процесс для запуска, он использует эти указатели, чтобы определить местоположение соответствующих таблиц страниц, и настраивает диспетчер памяти (MMU), использующий их. Даже если отсутствуют отдельные I- и D-пространства, процессы могут разделять программы (или, что иногда случается, библиотеки), но механизм совместного доступа усложняется.

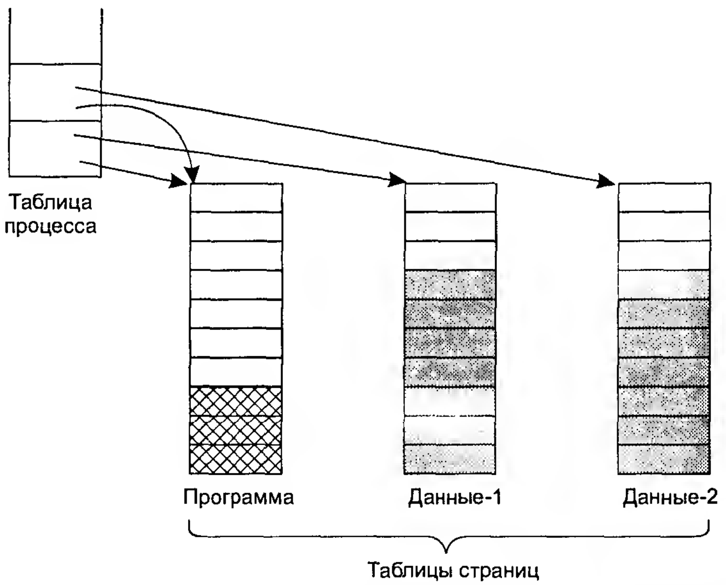


Рис. 4.29. Две процесса используют совместно одну и ту же программу, разделяя ее таблицу страниц

Когда два или более процессов совместно используют один программный код, возникает проблема разделения страниц. Предположим, что процессы *A* и *B* представляют собой дважды запущенный текстовый редактор и вместе используют его

страницы. Если планировщик решает удалить процесс *A* из памяти, выгрузка всех его страниц и заполнение пустых страничных блоков какой-либо другой программой приведет к тому, что процесс *B* вызовет массу страничных прерываний, чтобы вернуть назад эти страницы.

Когда процесс *A* завершает свою работу, очень важно знать, что страницы до сих пор используются, чтобы их дисковое пространство случайно не оказалось освобожденным. Поиск во всех таблицах страниц с целью проверки совместного использования страниц обычно слишком дорог, поэтому необходимы специальные структуры данных для отслеживания разделенных страниц, особенно если единицей совместного доступа является отдельная страница (или серия страниц), а не целая таблица страниц.

Совместное использование данных более сложно, чем совместное использование кода программы, но и оно возможно. В частности, в системе UNIX после системного вызова fork родительский и дочерний процессы обязаны совместно использовать и текст программы, и данные. В системах со страничной организацией памяти часто делается так: каждому из этих процессов дается своя собственная таблица страниц, но в них есть указатель на один и тот же набор страниц. Таким образом, не выполняется копирование страниц при вызове fork. Однако все страницы данных для обоих процессов отображаются как READ ONLY (только для чтения).

Пока оба процесса только читают свои данные, не модифицируя их, это состояние может не изменяться. Как только один из двух процессов обновляет слово в памяти, нарушение защиты «только для чтения» вызывает прерывание, передающее управление операционной системе. При этом создается копия страницы, и теперь каждый процесс имеет свой собственный персональный экземпляр страницы. Для обеих копий разрешается и чтение, и запись, поэтому последующие записи в любую из двух страниц происходят без прерываний. В результате работы такой системы те страницы, которые никогда не модифицируются (включая все страницы с текстом программы), не должны копироваться. Необходимо дублировать только страницы, содержащие фактически изменяемые данные. Такой подход, называемый **копированием при записи**, повышает производительность путем уменьшения количества операций дублирования.

Политика очистки страниц

Подкачка страниц работает лучше, когда в системе существует достаточное количество свободных страничных блоков, которые можно затребовать при страничном прерывании. Если все страничные блоки заполнены и, более того, изменялись, перед загрузкой новой страницы сначала нужно записать старую на диск. Чтобы обеспечить обильный запас свободных блоков, во многих системах со страничной организацией памяти работает фоновый процесс, называемый **страничным демоном**, который большую часть времени спит, но периодически просыпается и проверяет состояние памяти. Если свободно слишком мало блоков, страничный демон начинает выбирать страницы для удаления их из памяти, используя определенный алгоритм замещения. Если эти страницы изменялись со времени загрузки, они записываются на диск.

Так или иначе, запоминается прежнее содержимое страницы. В случае если одна из выгруженных страниц требуется снова еще до того, как ее блок был перезаписан, ее можно вернуть назад, удалив из пула свободных страничных блоков. Сохранение запаса страничных блоков в результате дает лучшую производительность, чем использование всей памяти и затем поиск блока в тот момент, когда он запрашивается. По крайней мере, страничный демон гарантирует, что все свободные блоки являются чистыми, значит, когда они требуются, их не нужно спешно записывать на диск.

Осуществить эту стратегию очистки страниц можно, например, с помощью часов с двумя стрелками. Передняя (длинная) стрелка контролируется страничным демоном. Когда она указывает на «грязную» страницу, копия страницы на диске обновляется, а стрелка сдвигается на позицию. Когда она направлена на чистую страницу, она просто сдвигается вперед. Задняя (короткая) стрелка используется для замещения страниц, как в стандартном алгоритме «часы». Только теперь возрастает вероятность попадания короткой стрелки на чистую страницу благодаря работе страничного демона.

Интерфейс виртуальной памяти

До сих пор в наших рассуждениях предполагалось, что виртуальная память прозрачна для процессов и программистов, то есть все, что они видят — это огромное виртуальное адресное пространство на компьютере с небольшой (или меньшей) физической памятью. Обычно это так и происходит, но в некоторых прогрессивных системах программистам предоставлен определенный контроль над картой памяти, который они могут использовать для улучшения поведения программы нетрадиционными способами. В этом разделе мы кратко рассмотрим некоторые из них.

Одной из причин предоставления программистам контроля над картой памяти является желание позволить двум и более процессам совместно использовать одну и ту же память. Если программисты сами будут давать названия областям памяти, один процесс сможет дать другому процессу имя области памяти, так что этот второй процесс также сможет ей пользоваться. Если два (или больше) процессов разделяют страницы памяти, становится реальной высокая пропускная способность совместного доступа — один процесс пишет в разделяемую память, а другой читает из нее.

Совместное использование страниц может также использоваться для реализации высокопроизводительных систем передачи сообщений. Когда передается сообщение, данные обычно копируются из одного адресного пространства в другое, что имеет значительную стоимость. Если процессы могут управлять своей картой страниц, можно передавать сообщения с помощью посылающего процесса, убирающего из карты страницу (страницы), содержащую сообщение, и принимающего процесса, вносящего ее (их) в карту. При этом должны копироваться только имена страниц вместо всех данных.

Еще одна современная техника управления памятью носит название **распределенной памяти совместного доступа** [114, 204, 205, 369]. Она основана на том, чтобы позволить нескольким процессам в сети совместно использовать набор

страниц, возможно (но не обязательно) как единственное разделяемое линейное адресное пространство. Когда процесс обращается к странице, не отображаемой в данный момент, он вызывает страничное прерывание. Обработчик страничных прерываний, который может находиться в ядре или в пользовательском пространстве, определяет машину, содержащую страницу, и посылает ей сообщение с просьбой выгрузить страницу и послать ее по сети. Когда страница прибывает, она попадает в карту и прерванная команда перезапускается. Мы будем более детально изучать распределенную память с совместным доступом в главе 8.

Вопросы реализации

Разработчики систем виртуальной памяти должны сделать выбор между основными теоретическими алгоритмами: «вторая попытка» или «старение», локальное распределение страниц или глобальное, предоставление страниц по запросу или опережающая подкачка страниц. Но они также должны быть осведомлены о количестве проблем, возникающих при практической реализации каждого варианта. В этом разделе мы рассмотрим несколько наиболее общих вопросов и некоторые из их решений.

Участие операционной системы в процессе подкачки страниц

Можно выделить четыре ситуации, в которых операционной системе приходится выполнять работу, относящуюся к страничной подкачке: создание процесса, выполнение процесса, страничное прерывание и завершение процесса. Сейчас мы кратко рассмотрим каждую из этих ситуаций по очереди, чтобы понять, что должно быть сделано в каждом случае.

При создании нового процесса в системе со страничной организацией памяти операционная система должна определить, насколько будут велики программа и данные к ней (в исходном состоянии), и создать для них таблицу страниц. Для таблицы страниц должно быть предоставлено пространство в памяти, и ее нужно проинициализировать. Таблица страниц не обязана быть резидентной в то время, когда процесс выгружается на диск, но она должна быть в памяти, пока процесс работает. Кроме того, в области подкачки на диске должно быть выделено пространство, чтобы в момент выгрузки страницы на диск было бы место, куда ее можно поместить. Область подкачки также нужно инициализировать текстом программы и данными, чтобы, когда новый процесс запустится, вызывая страничные прерывания, страницы можно было подгрузить с диска. Некоторые системы подкачивают страницы текста программы прямо из исполняемого файла, таким образом, экономя время инициализации и место на диске. Наконец, информация о таблице страниц и области подкачки на диске должна быть записана в таблицу процесса.

Когда процесс планируется для исполнения, для нового процесса требуется сброс диспетчера памяти (MMU), а содержимое буфера быстрого преобразования

адреса (TLB) должно быть очищено, чтобы избавиться от следов предыдущего процесса. Таблицу страниц нового процесса нужно сделать текущей, обычно это производится путем копирования ее или указателя на нее в некоторый аппаратный регистр (регистры). Часть страниц процесса или они все могут быть считаны в память, чтобы уменьшить изначальное количество страничных прерываний.

Когда происходит страничное прерывание, операционная система должна прочитать аппаратные регистры, чтобы определить, какой виртуальный адрес вызвал ошибку. Из полученной информации она должна вычислить, какая требуется страница, и определить ее местоположение на диске. Затем операционной системе нужно найти доступный страничный блок для размещения новой страницы, при необходимости она выгружает какую-либо старую страницу. После этого операционная система должна считать требуемую страницу в страничный блок. И наконец, она должна вернуть в предыдущее состояние счетчик команд, чтобы тот указывал на вызвавшую прерывание инструкцию, и запустить эту команду заново.

Когда процесс завершается, операционная система должна освободить его таблицу страниц, его страницы и дисковое пространство, которое занимают страницы, когда они находятся на диске. Если некоторые из страниц разделяются между несколькими процессами, страницы в памяти и на диске могут быть освобождены только тогда, когда окончит работу последний использующий их процесс.

Обработка страничного прерывания

Наконец мы подошли к моменту, когда можно более детально описать, что же происходит при страничном прерывании. Последовательность действий следующая:

1. Аппаратное обеспечение переключает систему в режим ядра, сохраняя счетчик команд в стеке. На большинстве машин в специальных регистрах процессора сохраняется некоторая информация о состоянии текущей инструкции.
2. Запускается написанная на ассемблере программа, сохраняющая основные регистры и другую изменяющуюся информацию, защищая ее от разрушения операционной системой. Эта программа вызывает операционную систему как процедуру.
3. Операционная система обнаруживает, что произошло страничное прерывание, и пытается найти необходимую виртуальную страницу. Часто требуемую информацию содержит один из аппаратных регистров. Если нет, операционная система должна достать из стека счетчик команд, выбрать инструкцию и программно проанализировать ее, чтобы определить, что она делала в тот момент, когда случилась ошибка.
4. Как только становится известен виртуальный адрес, вызвавший прерывание, система проверяет, имеет ли силу этот адрес и согласуется ли защита с доступом. Если нет, то процессу посылается сигнал или процесс уничтожается. Если адрес действителен и не произошло ошибки защиты, система проверяет наличие свободных страничных блоков. Если свободных блоков нет, запускается алгоритм замещения страниц, выбирающий жертву.

5. Если выбранный страничный блок «грязный», страница заносится в график записи на диск и происходит переключение контекста, приостанавливающее вызвавший прерывание процесс и позволяющее работать другому процессу до тех пор, пока не будет выполнен перенос страницы на диск. В любом случае блок отмечается как занятый, чтобы предотвратить его использование в других целях.
6. Как только страничный блок очищается (или немедленно, или после записи на диск), операционная система ищет адрес на диске, где находится требуемая страница, и планирует дисковую операцию для ее переноса в память. Во время загрузки страницы процесс, вызвавший прерывание, все еще приостановлен и выполняется другой пользовательский процесс, если такой доступен.
7. Когда дисковое прерывание отмечает, что страница поступила в память, обновляется таблица страниц, отражая ее позицию, а блок помечается, как находящийся в нормальном состоянии.
8. Прерванная команда возвращается к тому состоянию, с которого она начиналась, и значение счетчика команд приостановленного процесса (в стеке или в системной ячейке памяти) корректируется так, чтобы указывать на эту команду.
9. Прерванный процесс вносится в график, и операционная система возвращает управление ассемблерной процедуре, вызывавшей ее.
10. Эта процедура перезагружает регистры и другую информацию о состоянии и возвращает управление в пользовательское пространство для продолжения выполнения пользовательской программы, как если бы никакого прерывания не происходило.

Перезапуск прерванной команды процессора

Когда программа обращается к странице, которой нет в памяти, команда процессора, вызвавшая прерывание, останавливается на полпути, и происходит прерывание с передачей управления операционной системе. После того как операционная система перенесла в память необходимую страницу, она должна перезапустить команду, вызвавшую прерывание. Это проще сказать, чем сделать.

Чтобы увидеть суть данной проблемы в черном цвете, рассмотрим центральный процессор, в который поступила команда с двумя адресами, например процессор Motorola 680x0, широко используемый во встроенных системах. Команда

```
MOVE.L #6(A1),2(A0)
```

занимает 6 байт (рис. 4.30). Для того чтобы перезапустить ее, операционная система должна определить, где находится первый байт команды. Значение счетчика команд во время прерывания зависит от того, какой операнд вызвал ошибку, и от реализации микрокода процессора.

На рис. 4.30 у нас есть команда, начинающаяся с адреса 1000, которая совершает три обращения к памяти: само слово команды и два сдвига для операндов. В зависимости от того, какое из этих трех обращений к памяти вызывает странич-

ное прерывание, счетчик команд может принять значение 1000, 1002 или 1004 во время прерывания. Зачастую операционная система не в силах однозначно определить, где начиналась команда. Если в момент прерывания счетчик команд равен 1002, у операционной системы нет способа определить, является ли слово по адресу 1002 адресом памяти, связанным с командой по адресу 1000 (то есть операндом) или кодом операции самой команды.

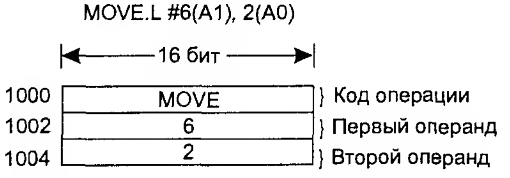


Рис. 4.30. Команда, вызвавшая страничное прерывание

Какой бы неприятной ни была данная проблема, могло бы быть гораздо хуже. У процессора 680x0 есть очень удобные для программистов автоинкрементные режимы адресации, а это означает, что побочным эффектом выполнения команды является увеличение (или уменьшение) одного или двух регистров. Инструкции, использующие автоинкрементный режим, также могут прерываться. В зависимости от деталей микропрограммы приращение может быть сделано до обращения к памяти, и в этом случае операционная система должна уменьшить регистр программно перед тем, как запустить команду заново. Автоматическое приращение может выполняться и после обращения к памяти, в этом случае оно еще не было сделано в момент прерывания и не должно отменяться операционной системой. Также существует режим автоматического уменьшения на единицу, и он вызывает ту же самую проблему. Точные детали того, выполнялось автоувеличение или автоуменьшение регистра до или после обращений к памяти, могут различаться для разных команд и для различных моделей процессоров.

К счастью, на некоторых машинах разработчики центральных процессоров предусматривают решение этой проблемы обычно в форме скрытых внутренних регистров, в которые копируется счетчик команд перед выполнением каждой инструкции. Такие машины также могут иметь второй регистр, говорящий о том, какой из регистров и на сколько уже был увеличен или уменьшен. Обладая этой информацией, операционная система в силах однозначно отменить все эффекты прерванной инструкции, так что потом ее можно целиком запустить заново. Если эта информация недоступна, операционной системе придется покрутиться, чтобы понять, что же случилось и как это исправить. Видимо, разработчики аппаратуры не смогли решить эту проблему, поэтому они опустили руки и предоставили возможность разбираться с этим разработчикам операционных систем. Славные парни.

Блокирование страниц в памяти

Хотя в этой главе мы практически не вспоминали о вводе-выводе, тот факт, что компьютер имеет виртуальную память, вовсе не означает, что ввод-вывод отсутствует. Виртуальная память и ввод-вывод незаметно взаимодействуют друг с другом. Рассмотрим процесс, который только что сделал системный вызов, чтобы считать

информацию из некоторого файла или устройства в буфер внутри своего адресного пространства. Пока процесс ожидает завершения операции ввода-вывода, он приостанавливается, предоставляя возможность работы другому процессу. Этот другой процесс вызывает страничное прерывание.

Если алгоритм подкачки страниц — глобальный, существует маленький, но не равный нулю шанс, что страница, содержащая буфер ввода-вывода, будет выбрана для удаления из памяти. Если устройство ввода-вывода в данный момент находится в процессе выполнения прямой передачи данных в эту страницу, ее удаление станет причиной записи части данных в буфер, которому они принадлежат, а часть данных запишется в заново загруженную страницу. Одно решение этой проблемы заключается в том, чтобы блокировать страницы, занятые вводом-выводом, так чтобы они не удалялись из памяти. Блокирование страницы часто называется **пришпиливанием** (pinning) ее в памяти. Другое решение — это сначала выполнить весь ввод-вывод в буферы ядра, а затем копировать данные в пользовательские страницы.

Хранение страничной памяти на диске

В ходе рассмотрения алгоритмов замещения страниц мы видели, как выбирается страница для удаления. Мы почти ничего не сказали о том, в какое место на диске она помещается после выгрузки из памяти. Теперь настало время описать некоторые из проблем, связанных с управлением дисками.

Простейший алгоритм для распределения страничного пространства на диске заключается в поддержке специальной области подкачки (свопинга) на диске. При загрузке системы эта область является пустой и представляется в памяти единой записью, имеющей свой начальный адрес и размер. Когда запускается первый процесс, резервируется участок области подкачки размером с этот процесс, а остальная область уменьшается ровно на это количество. Как только запускаются новые процессы, им предоставляются участки области подкачки, равные по размеру их образам памяти. Как только они завершаются, их дисковое пространство освобождается. Область подкачки управляется как список свободных участков.

С каждым процессом связывается адрес его области подкачки на диске, который хранится в таблице процесса. Вычисление адреса для записи страницы становится простым: всего лишь прибавляется смещение страницы внутри виртуального адресного пространства к адресу начала области подкачки. Однако перед тем, как процесс может начать работу, область подкачки должна быть инициализирована. Это можно сделать, копируя полный образ процесса в область подкачки так, чтобы его по необходимости можно было *переносить в память*. Второй способ: загрузить весь процесс в память и позволить ему постранично *выгружаться на диск*, также когда это требуется.

Но в такой простой модели есть одна проблема: процессы могут увеличиваться в размере после запуска. Хотя текст программы обычно фиксирован, иногда может расти область данных и всегда может увеличиваться стек. Следовательно, стратегия, заключающаяся в резервировании отдельных областей подкачки для

текста, данных и стека и позволении каждой из этих областей состоять более чем из одного участка на диске, может оказаться более удачной.

Другая крайность состоит в том, чтобы, наоборот, ничего не размещать заранее; предоставлять пространство на диске для каждой страницы, когда она выгружается на диск, и освобождать это место, когда она подкачивается обратно в память. При таком подходе процессы в памяти не привязаны к какому-либо пространству подкачки на диске. Его недостаток состоит в том, что адрес на диске, необходимый в памяти, отслеживается для каждой страницы на диске. Другими словами, для каждого процесса должна поддерживаться таблица, содержащая местоположение каждой страницы на диске. Эти два альтернативных варианта показаны на рис. 4.31.

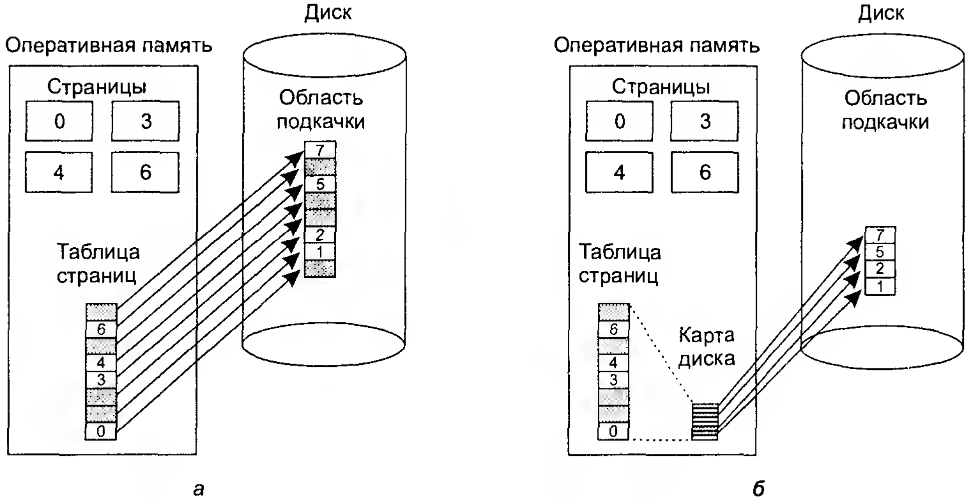


Рис. 4.31. Постраничная подкачка в статическую область свопинга (*а*); динамическое резервирование страниц (*б*)

На рис. 4.31, *а* продемонстрирована таблица страниц с восемью страницами. Страницы 0, 3, 4 и 6 находятся в оперативной памяти. Страницы 1, 2, 5 и 7 хранятся на диске. Область подкачки на диске равна по размеру виртуальному адресному пространству процесса (восемь страниц), причем у каждой страницы есть фиксированное место, куда она записывается при удалении из оперативной памяти. Вычисление ее адреса требует информации только о том, где начинается область страничной подкачки процесса на диске, так как страницы хранятся в ней последовательно в порядке своих виртуальных номеров. Страницы, находящиеся в памяти, всегда имеют дубликат на диске, но эта копия может быть устаревшей, если страница была изменена с момента загрузки в память.

На рис. 4.31, *б* страницы не имеют фиксированных адресов на диске. Когда страница выгружается из памяти, на ходу выбирается пустая страница на диске, и соответственно обновляется карта диска (с ячейками для каждого дискового адреса, соответствующего одной виртуальной странице). У страниц, находящихся в памяти, нет копий на диске. Их записи в карте диска содержат ошибочный дисковый адрес или бит, маркирующий их как не использующиеся.

Разделение политики и механизма

Важным инструментом для управления любой комплексной системой является отделение политики от механизма. Этот принцип можно применить к управлению памятью, осуществив работу большей части управления как процесса на пользовательском уровне. Такое разделение впервые было осуществлено в системах Mach [366] и MINIX [322]. Дальнейшие рассуждения приближенно базируются на работе системы Mach.

Простой пример того, как могут быть разделены стратегия и механизм, показан на рис. 4.32. Здесь управление памятью делится на три части:

1. Низкоуровневый драйвер диспетчера памяти (MMU).
2. Обработчик страничных прерываний, являющийся частью ядра.
3. Внешний обработчик страниц, работающий в пользовательском пространстве.

Все детали работы диспетчера памяти инкапсулированы в драйвере MMU. Он представляет собой машинозависимую программу и должен переписываться для каждой новой платформы, на которую переносится операционная система. Обработчик страничных прерываний является программой, не зависящей от машины, и содержит большую часть технических средств для страничной подкачки. Политика в основном определяется внешним обработчиком страниц, работающим в пользовательском пространстве.

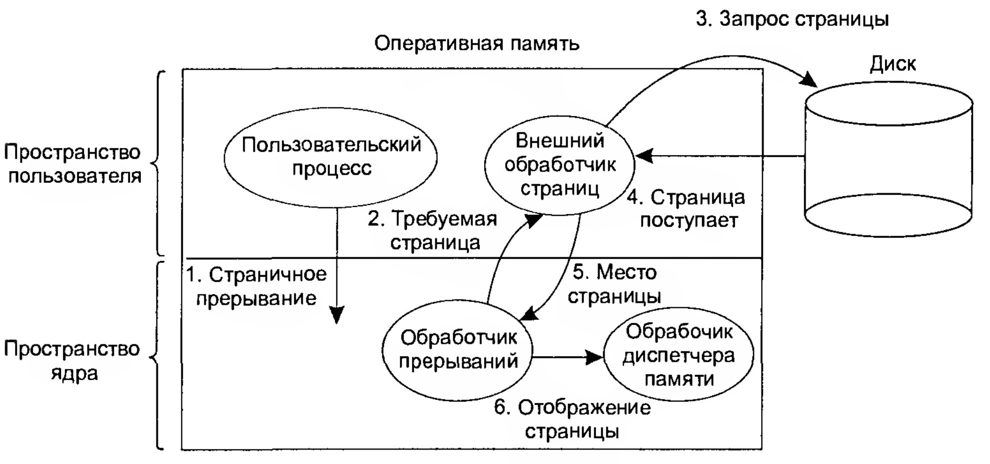


Рис. 4.32. Обработка страничного прерывания с внешним обработчиком страниц

При запуске процесса извещается внешний обработчик страниц для того, чтобы установить карту страниц процесса и в случае необходимости предоставить место на диске для резервного хранения. Во время работы процесс может отображать новые объекты в свое адресное пространство, о чем опять уведомляется внешний обработчик страниц.

Как только процесс начинает работу, он может вызвать страничное прерывание. Обработчик прерываний вычисляет, какая страница требуется, и посылает сообщение внешнему обработчику страниц, уведомляя его, в чем заключается

проблема. Тогда внешний обработчик страниц считывает нужную страницу с диска и копирует ее в часть его собственного адресного пространства. Затем он говорит обработчику прерываний, где находится страница. Обработчик прерываний убирает отображение страницы из адресного пространства внешнего обработчика страниц и просит драйвер MMU поместить ее в нужное место в пользовательском адресном пространстве. После этого можно перезапускать пользовательский процесс.

Эта реализация оставляет открытым вопрос о размещении алгоритма замещения страниц. Возможно, покажется более удачным расположить его во внешнем обработчике страниц, но тогда возникают некоторые проблемы. Из них принципиально то, что внешний обработчик страниц не имеет доступа к битам *R* и *M* для всех страниц. А эти биты играют важнейшую роль во многих алгоритмах страничной подкачки. Таким образом, или необходим какой-либо механизм для передачи содержимого битов внешнему обработчику страниц, или же алгоритм замещения страниц должен находиться в ядре. Во втором случае обработчик прерываний сообщает внешнему обработчику, какую страницу он выбрал для удаления, и предоставляет данные или путем отображения их в адресное пространство внешнего обработчика страниц, или включая их в сообщение. В любом варианте внешний обработчик страниц пишет данные на диск.

Основным преимуществом такой реализации является большая модульность программы и большая гибкость. Основной недостаток заключается в дополнительных расходах на несколько переходов границы «пользовательское пространство — ядро» и на различные сообщения, пересылаемые между частями системы. В настоящее время этот вопрос вызывает массу дискуссий, но поскольку компьютеры приобретают все большую скорость, а программы становятся все более и более сложными, вероятно, в итоге большинство разработчиков согласятся принести в жертву некоторый процент производительности для того, чтобы иметь более надежное программное обеспечение.

Сегментация

Обсуждавшаяся до сих пор виртуальная память представляет собой одномерное пространство, потому что виртуальные адреса идут один за другим от 0 до некоторого максимума. Для многих задач наличие двух и более отдельных виртуальных адресных пространств может оказаться намного лучше, чем всего одно. Например, у компилятора есть много таблиц, которые формируются по мере трансляции, возможно, включая в себя:

1. Исходный текст, сохраненный для печати листинга (в пакетных системах).
2. Символьную таблицу, содержащую имена и атрибуты переменных.
3. Таблицу, содержащую все используемые константы: целые и с плавающей точкой.
4. Дерево грамматического разбора, содержащее синтаксический анализ программы.
5. Стек, используемый для процедурных вызовов внутри компилятора.

Во время компиляции каждая из первых четырех таблиц непрерывно растет. Последняя таблица при компиляции непредсказуемо увеличивается или уменьшается. В одномерной памяти эти пять таблиц должны были бы размещаться в смежных частях виртуального адресного пространства, как на рис. 4.33.

Рассмотрим, что происходит, если программа имеет исключительно большое число переменных, но нормальное количество всего остального. Участок адресного пространства, предоставленный для таблицы кодировки символов, может заполниться, но в других таблицах, скорее всего, останется пустым множество ячеек. Конечно, компилятор может просто создать сообщение о том, что компиляция не может продолжаться вследствие слишком большого количества переменных, но нам кажется, что такое решение проблемы не спортивно, когда в других таблицах осталась масса неиспользованного места.

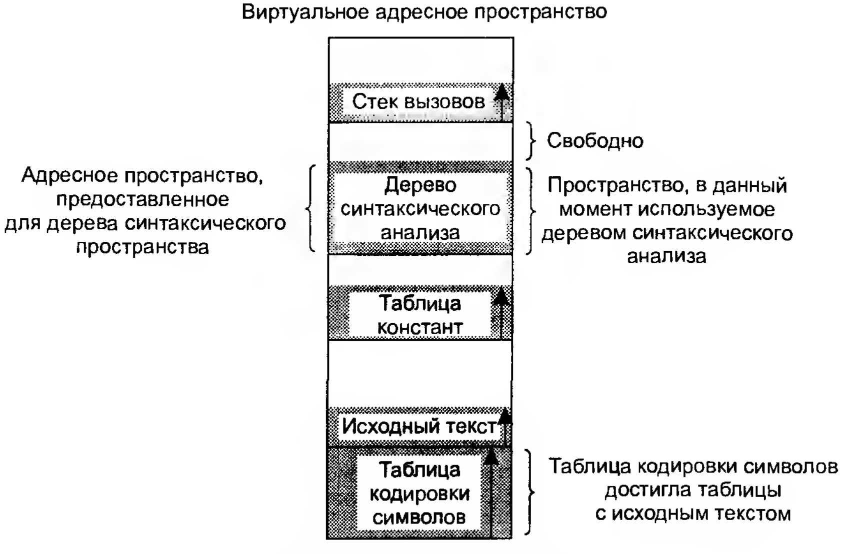


Рис. 4.33. В одномерном адресном пространстве при росте таблиц одна может упереться в другую

При другом варианте можно поиграть в Робин Гуда, забирая пространство из таблиц с излишеством ячеек и передавая их таблицам с их недостатком. Такая перетасовка реализуема, но она аналогична управлению собственными оверлеями, что представляет собой маленькое неудобство в лучшем случае и большое количество скучной и неоплачиваемой работы в худшем случае.

На самом деле необходим метод, освобождающий программиста от управления расширяющимися и сокращающимися таблицами тем же самым способом, которым виртуальная память устраняет беспокойство организации программ с оверлеями.

Простое и предельно общее решение заключается в том, чтобы обеспечить машину множеством полностью независимых адресных пространств, называемых **сегментами**. Каждый сегмент содержит линейную последовательность адресов от 0 до некоторого максимума. Длина каждого сегмента может быть любой от нуля до разрешенного максимума. Различные сегменты могут быть различной длины. Более

того, длины сегментов могут изменяться во время выполнения. Длина сегмента стека может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться при выборке данных из стека.

Поскольку каждый сегмент составляет отдельное адресное пространство, разные сегменты могут расти или сокращаться независимо друг от друга. Если стек, находящийся в определенном сегменте, нуждается в большем количестве адресного пространства для роста, он может получить его, потому что в его адресном пространстве нет больше ничего, с чем можно столкнуться. Конечно, сегмент может заполниться, но сегменты обычно очень большие, поэтому такие инциденты редки. Чтобы определить адрес в такой сегментированной или двумерной памяти, программа должна указать адрес, состоящий из двух частей: номер сегмента и адрес внутри сегмента.

Рисунок 4.34 иллюстрирует сегментированную память, использующуюся для обсуждавшихся ранее таблиц компилятора. Здесь показаны пять независимых сегментов.

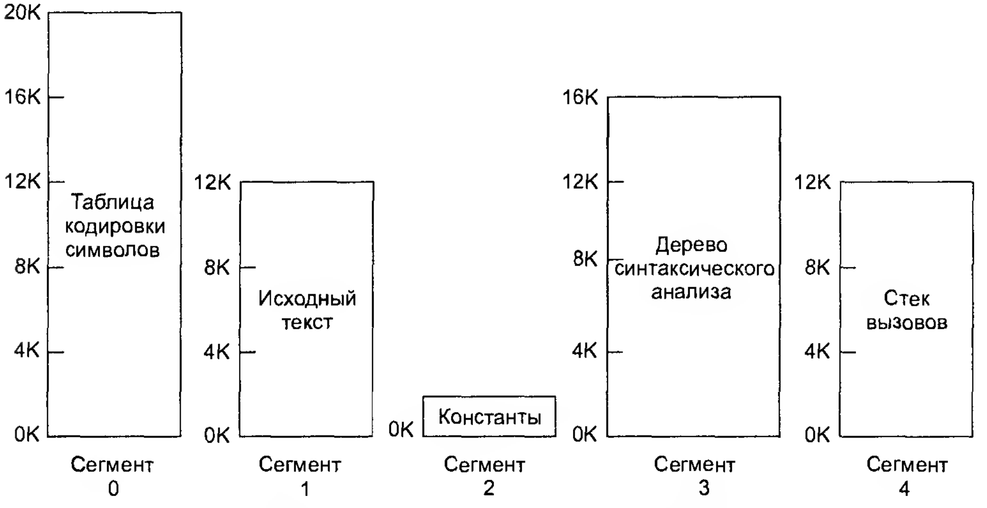


Рис. 4.34. Сегментированная память позволяет каждой таблице расти или уменьшаться независимо от других таблиц

Стоит подчеркнуть, что сегмент — это логический объект, о чем программист знает и поэтому использует его как логический объект. Сегмент может иметь в составе процедуру, массив, стек или набор скалярных переменных, но обычно он не содержит смеси различных типов.

Помимо простоты управления увеличивающимися или сокращающимися структурами данных, сегментированная память обладает и другими преимуществами. Если каждая процедура занимает отдельный сегмент и адрес 0 — это ее начальный адрес, компоновка отдельно скомпилированных процедур происходит намного проще. После того как все процедуры, составляющие программу, будут скомпилированы и скомпонованы, для адресации слова 0 (начальной точки) обращение к процедуре в сегменте *n* будет использовать адрес, состоящий из двух частей (*n*, 0).

Если потом процедура в сегменте *n* модифицируется и компилируется заново, не нужно изменять другие процедуры (потому что начальный адрес остался тем же), даже если новая версия больше предыдущей. В одномерной памяти процедуры упакованы одна к другой и между ними нет свободного адресного пространства. В результате изменение размера одной процедуры может повлиять на начальный адрес другой, не имеющей отношения к первой процедуре. Это, в свою очередь, требует модификации всех процедур, вызывающих любую из передвинутых процедур, чтобы поместить в них новые начальные адреса. Если программа содержит сотни процедур, такой процесс очень дорог.

Сегментация также облегчает совместное использование процедур и данных несколькими процессами. Общим примером является **библиотека совместного доступа**. Современные рабочие станции, работающие с передовыми оконными системами, часто имеют крайне большие графические библиотеки, являющиеся составляющими практически каждой программы. В сегментированных системах графические библиотеки могут располагаться в отдельном сегменте и совместно использоваться несколькими процессами, что устраняет необходимость их присутствия в адресном пространстве каждого процесса. В принципе в системах с чистой страничной организацией памяти также можно иметь совместно используемые библиотеки, но это намного сложнее в реализации. Поэтому такие системы предоставляют совместный доступ путем моделирования сегментации.

Поскольку каждый сегмент формирует логический объект (такой как процедура, массив или стек), с которым общается программист, у различных сегментов могут быть разные виды защиты. Сегмент процедуры может быть определен как только исполняемый, что запрещает попытки чтения из него или сохранения в него. Для массива чисел с плавающей точкой можно разрешить режим доступа чтение/запись, но не исполнение, чтобы отлавливать попытки передачи управления по адресам, на которых располагается массив. Такая защита полезна при обнаружении ошибок программирования.

Вы должны попытаться понять, почему защита имеет смысл в сегментированной памяти, а не в одномерной страничной памяти. В сегментированной памяти пользователь осведомлен о том, что представляет собой каждый сегмент. В обычном случае сегмент не может содержать, например, и процедуру и стек, а только либо первое, либо второе. Так как каждый сегмент содержит только один тип объектов, он может иметь защиту, соответствующую этому конкретному типу. Страничная организация памяти и сегментация сравниваются в табл. 4.3.

Содержимое страниц в известной степени случайно. Программист не осведомлен даже о том факте, что происходит страничная подкачка. Хотя добавление нескольких битов в каждую запись таблицы страниц для определения разрешенного доступа в принципе возможно, но, чтобы использовать это свойство, программист должен был бы отслеживать, где находятся границы страниц в его адресном пространстве. Это представляет собой в точности тот вид администрирования, для устранения которого была придумана страничная подкачка. Поскольку пользователь сегментированной памяти имеет дело с иллюзией постоянного нахождения всех сегментов в оперативной памяти — то есть он может адресоваться к ним так, как будто они существуют, — он может защищать сегменты по отдельности, не заботясь об управлении их загрузкой в память.

Таблица 4.3. Сравнение страничной организации памяти и сегментации

Вопрос	Страничная память	Сегментация
Нужно ли программисту знать о том, что используется эта техника?	Нет	Да
Сколько в системе линейных адресных пространств?	1	Много
Может ли суммарное адресное пространство превышать размеры физической памяти?	Да	Да
Возможно ли разделение процедур и данных, а также раздельная защита для них?	Нет	Да
Легко ли размещаются таблицы с непостоянными размерами?	Нет	Да
Облегчен ли совместный доступ пользователей к процедурам?	Нет	Да
Зачем была придумана эта техника?	Чтобы получить большое линейное адресное пространство без дополнительных затрат на физическую память	Для возможности разбиения программ и данных на логически независимые адресные пространства, облегчения совместного доступа и защиты

Реализация сегментации

Реализация сегментации существенно отличается от страничной организации памяти: страницы имеют фиксированный размер, а сегменты — нет. На рис. 4.35, *а* показан пример физической памяти, изначально содержащей пять сегментов. Теперь рассмотрим, что произойдет, если удаляется сегмент 1, а на его место помещается сегмент 7 меньшего размера. Мы получим конфигурацию памяти, изображенную на рис. 4.35, *б*. Между сегментом 7 и сегментом 2 расположена неиспользуемая область, то есть дыра. Затем сегмент 4 замещается сегментом 5 (см. рис. 4.35, *в*), а сегмент 3 заменяется сегментом 6, как на рис. 4.35, *г*. После того как система поработает какое-то время, память разделится на некоторое количество участков, часть из которых содержит сегменты, а остальные свободны. Этот феномен разделения памяти на маленькие свободные участки, которые сложно использовать, называется **поклеточной разбивкой** или **внешней фрагментацией**. С внешней фрагментацией можно бороться с помощью уплотнения, как показано на рис. 4.35, *д*.

Сегментация с использованием страниц: система MULTICS

При большом размере сегментов может быть неудобно или даже невозможно хранить их в оперативной памяти целиком. Это приводит к идее их страничной организации, чтобы поблизости находились только те страницы, которые на самом деле нужны. Страничные сегменты поддерживались несколькими важными для нас системами. В этом разделе мы будем описывать первую из них: систему MULTICS. В следующем разделе мы обратимся к более современной системе Intel Pentium.

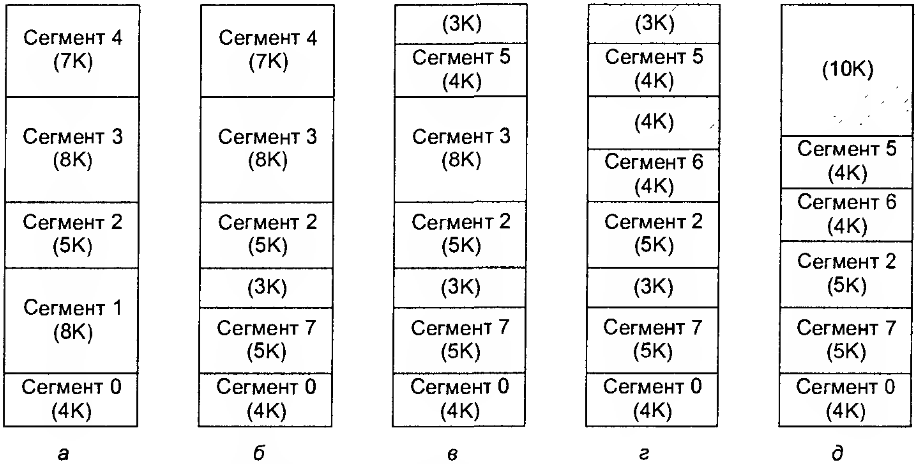


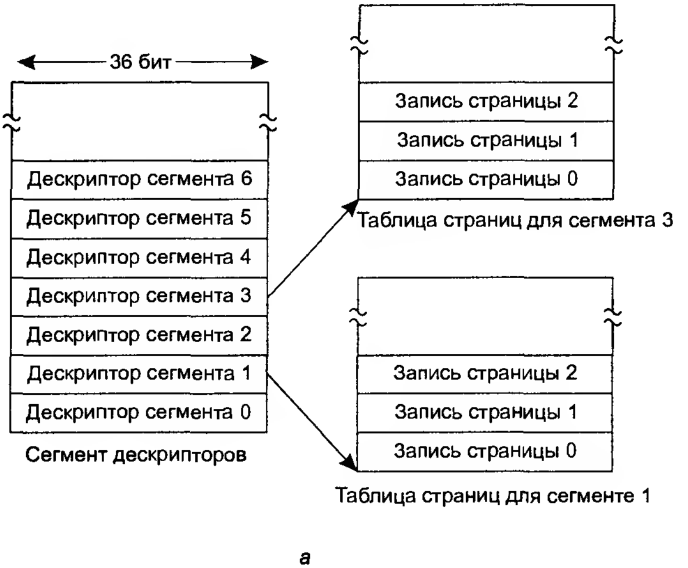
Рис. 4.35. Развитие внешней фрагментации (*а—г*); устранение фрагментации с помощью уплотнения (*д*)

Система MULTICS работала на компьютерах Honeywell 6000 и их потомках и обеспечивала каждую программу виртуальной памятью размером вплоть до 2^{18} сегментов (более 250 000), каждый из которых мог быть до 65 536 (36-разрядных) слов длиной. Чтобы осуществить это, разработчики системы MULTICS решили трактовать каждый сегмент как виртуальную память и разбить его на страницы, комбинируя преимущества страничной организации памяти (постоянный размер страницы и отсутствие необходимости хранения целого сегмента в памяти, если используется только его часть) с преимуществом сегментации (облегчение программирования, модульности, защиты и совместного доступа).

Каждая программа в системе MULTICS имеет таблицу сегментов с одним дескриптором на сегмент. Так как записей в таблице потенциально больше четверти миллиона, таблица сегментов сама является сегментом и разбита на страницы. Дескриптор сегмента содержит индикатор того, находится ли сегмент в памяти или нет. Если какая-то часть сегмента присутствует в памяти, считается, что сегмент в памяти и его таблица страниц будет в памяти. Если сегмент находится в памяти, то его дескриптор содержит 18-разрядный указатель на его таблицу страниц (рис. 4.36, *а*). Поскольку физические адреса 24-разрядные, а страницы выстраиваются по 64-байтным границам (предполагается, что 6 бит низших разрядов адреса страницы — это 000000), необходимо только 18 бит в дескрипторе для хранения адреса таблицы страниц. Дескриптор также содержит размер сегмента, биты защиты и несколько других полей. Рисунок 4.36, *б* демонстрирует дескриптор сегмента в системе MULTICS. Адрес сегмента во вспомогательной памяти не находится в дескрипторе сегмента, но в другой таблице используется обработчиком сегментных прерываний.

Каждый сегмент представляет собой обыкновенное адресное пространство и разбит на страницы точно так же, как и несегментированная страничная память, описанная ранее в этой главе. Нормальный размер страницы равен 1024 словам (хотя несколько меньшие сегменты, используемые MULTICS, не разбиты на

страницы или же все-таки разделены на страницы по 64 слова, чтобы сохранить физическую память).



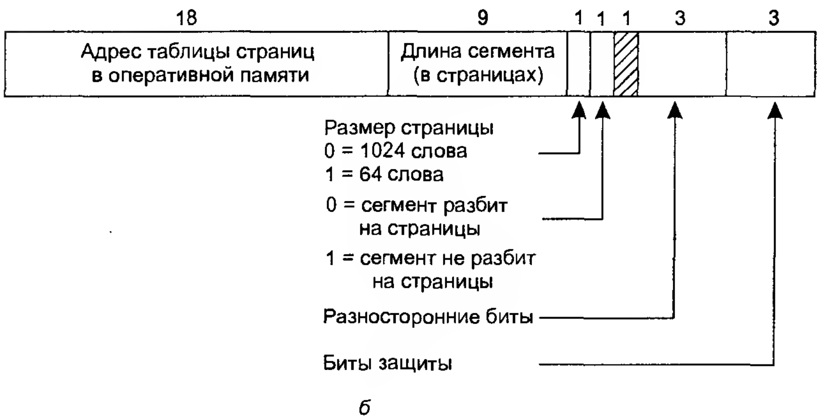


Рис. 4.36. Виртуальная память системы MULTICS: сегмент дескрипторов указывает на таблицы страниц (*а*); дескриптор сегмента (*б*). Числа означают длину полей

Адрес в системе MULTICS состоит из двух частей: сегмента и адреса внутри сегмента. Последний, в свою очередь, делится на номер страницы и слово внутри страницы, как показано на рис. 4.37. Когда происходит обращение к памяти, выполняется следующий алгоритм:

1. По номеру сегмента находится дескриптор сегмента.
2. Проверяется, находится ли таблица страниц сегмента в памяти. Если таблица страниц в памяти, определяется ее расположение. Если нет, вызывается сегментное прерывание. При нарушении защиты происходит прерывание.

3. Изучается запись в таблице страниц для запрашиваемой виртуальной страницы. Если страница не находится в памяти, происходит страничное прерывание. Если она в памяти, из записи таблицы страниц извлекается адрес начала страницы в оперативной памяти.
4. К адресу начала страницы прибавляется смещение, что дает в результате адрес в оперативной памяти, где расположено нужное слово.
5. Наконец, происходит чтение или сохранение.

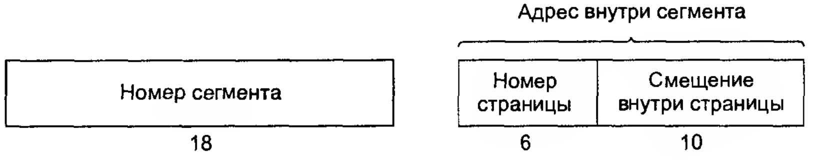


Рис. 4.37. 34-разрядный виртуальный адрес в системе MULTICS

Этот процесс продемонстрирован на рис. 4.38. Для простоты был опущен тот факт, что сегмент дескрипторов сам имеет страничное строение. Реально происходит следующее: регистр (основной регистр дескриптора) используется для определения расположения таблицы страниц сегмента дескрипторов, которая, в свою очередь, указывает на страницы сегмента дескрипторов. Как только дескриптор для необходимого сегмента найден, адресация продолжается, как показано на рис. 4.38.

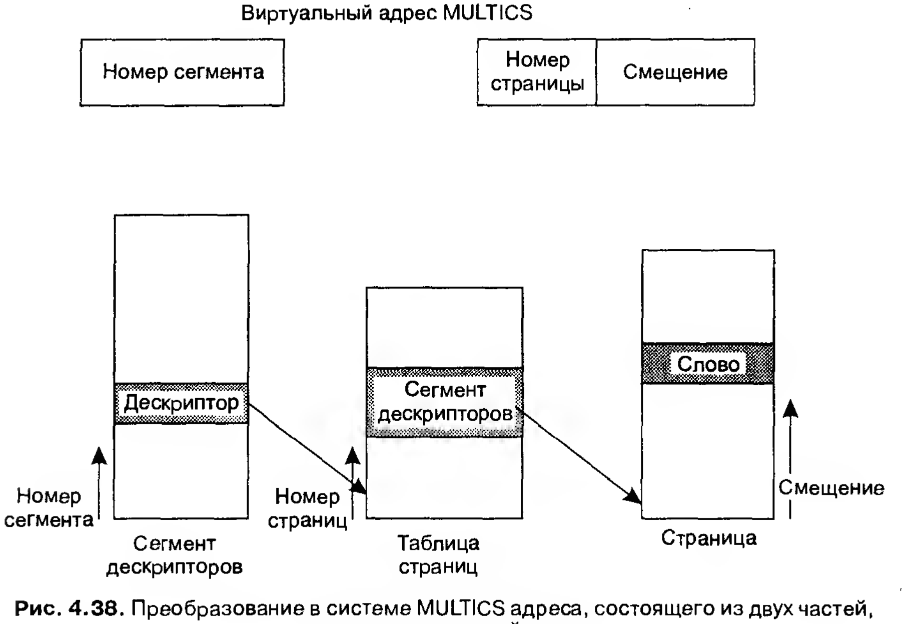


Рис. 4.38. Преобразование в системе MULTICS адреса, состоящего из двух частей, в адрес в оперативной памяти

Как вы, без сомнения, теперь догадались, если бы на практике предыдущий алгоритм выполнялся операционной системой для каждой команды процессора, работа программы была бы не слишком быстрой. В действительности аппаратура системы

MULTICS содержит высокоскоростной буфер быстрого преобразования адреса (TLB) размером в 16 слов, способный производить поиск параллельно по всем своим записям для заданного ключа. Это проиллюстрировано на рис. 4.39. Когда компьютер получает адрес, аппаратура адресации сначала проверяет наличие виртуального адреса в буфере TLB. Если это так, она получает номер страничного блока прямиком из буфера TLB и формирует фактический адрес слова, к которому происходит обращение, не выполняя поиск в сегменте дескрипторов или таблице страниц.

Поле сравнения					
Номер сегмента	Виртуальная страница	Страничный блок	Защита	Возраст	Эта запись используется?
4	1	7	Чтение/запись	13	1
6	0	2	Только чтение	10	1
12	3	1	Чтение/запись	2	1
					0
2	1	0	Только выполнение	7	1
2	2	12	Только выполнение	9	1

Рис. 4.39. Простейший вариант буфера быстрого преобразования адреса (TLB) в системе MULTICS. Существование двух размеров страниц делает фактическое строение буфера TLB более сложным

Адреса 16 наиболее часто использующихся страниц хранятся в буфере быстрого преобразования адреса. Программы, у которых рабочий набор меньше размера буфера TLB, будут хранить адреса всего рабочего набора в буфере TLB и, следовательно, будут работать эффективно. Если страница не находится в буфере быстрого преобразования адреса, фактически происходит обращение к дескриптору и таблице страниц, чтобы найти адрес страничного блока, и буфер TLB обновляется, чтобы включить эту страницу. При этом выгружается страница, не использовавшаяся дольше всего. Поле «возраста» хранит информацию о том, какая из записей использовалась наиболее давно. Причиной для применения буфера TLB служит обеспечиваемое им параллельное сравнение сегментов и номеров страниц всех записей.

Сегментация с использованием страниц: Intel Pentium

Виртуальная память на компьютере Pentium во многих отношениях аналогична памяти в системе MULTICS, включая наличие и сегментации, и страничной организации. В то время как система MULTICS имеет 256 К независимых сегментов, каждый до 64 К 36-разрядных слов, система Pentium поддерживает 16 К независимых сегментов, каждый до 1 млрд 32-разрядных слов. Хотя в последней системе меньше сегментов, их увеличенный размер намного более важен, так как программы редко нуждаются более чем в 1000 сегментах, но многим программам необходимы сегменты значительного размера.

Основа виртуальной памяти системы Pentium состоит их двух таблиц: **локальной таблицы дескрипторов LDT** (Local Descriptor Table) и **глобальной таблицы дескрипторов GDT** (Global Descriptor Table). У каждой программы есть своя собственная таблица LDT, но глобальная таблица дескрипторов одна, ее совместно используют все программы в компьютере. Таблица LDT описывает сегменты, локальные для каждой программы, включая ее код, данные, стек и т. д., тогда как таблица GDT несет информацию о системных сегментах, включая саму операционную систему.

Чтобы получить доступ к сегменту, программа системы Pentium сначала загружает селектор для этого сегмента в один из шести сегментных регистров машины. Во время выполнения регистр CS содержит селектор для сегмента кода команд, а регистр DS хранит селектор для сегмента данных. Каждый селектор представляет собой 16-разрядный номер (рис. 4.40).

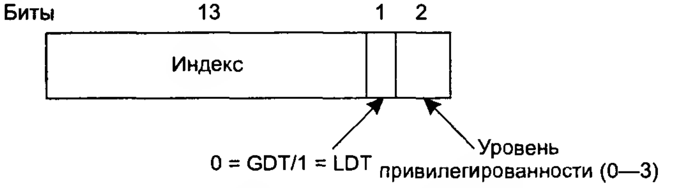


Рис. 4.40. Селектор в системе Pentium

Один из битов селектора несет информацию о том, является ли данный сегмент локальным или глобальным (то есть находится ли он в локальной или глобальной таблице дескрипторов). Следующие тринадцать битов определяют номер записи в таблице дескрипторов, поэтому эти таблицы ограничены: каждая содержит 8 К сегментных дескрипторов. Остальные 2 бита относятся к проблемам защиты и будут описаны позже. Дескриптор 0 является запрещенным. Его можно безопасно загрузить в сегментный регистр, чтобы обозначить, что сегментный регистр в данный момент недоступен. При попытке его использовать происходит прерывание.

Во время загрузки селектора в сегментный регистр соответствующий дескриптор извлекается из локальной или глобальной таблицы дескрипторов и сохраняется в микропрограммных регистрах, что обеспечивает к нему быстрый доступ. Дескриптор состоит из 8 байт, в которые входит базовый адрес сегмента, размер и другая информация (рис. 4.41).

Формат селектора искусно выбирался так, чтобы упростить определение местоположения дескриптора. Сначала выбирается локальная или глобальная таблица дескрипторов, основываясь на бите два селектора. Затем селектор копируется во внутренний рабочий регистр и три младших бита приравниваются к 0. Наконец, к нему прибавляется адрес одной из таблиц, чтобы получить прямой указатель на дескриптор. Например, селектор 72 ссылается на запись 9 в глобальной таблице дескрипторов, расположенную по адресу в таблице GDT+72.

Теперь проследим шаги, с помощью которых пара (селектор, смещение) преобразуется в физический адрес. Как только микропрограмма узнает, какой сегментный регистр используется, она может найти в своих внутренних регистрах полный дескриптор, соответствующий этому селектору. Если сегмент не существует (селектор равен 0) или в данный момент выгружен, возникает прерывание.

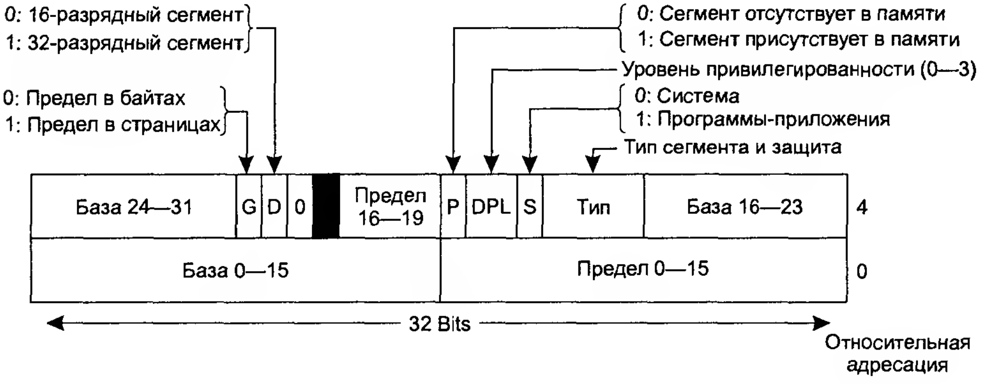


Рис. 4.41. Дескриптор программного сегмента в системе Pentium. Сегменты данных немного отличаются от программных сегментов

Затем она проверяет, выходит ли смещение за пределы сегмента, в случае чего также возникает прерывание. Логически в дескрипторе просто должно существовать 32-разрядное поле, дающее размер сегмента, но там доступны только 20 бит, поэтому используется другая схема. Если поле *Gbit* (granularity — глубина детализации) равно 0, поле *Limit* (предел) содержит точный размер сегмента, до 1 Мбайт. Если оно равно 1, поле *Limit* дает размер сегмента в страницах вместо байтов. Размер страницы в системе Pentium фиксирован на величине 4 Кбайт, поэтому 20 битов достаточно для сегментов размером до 2^{32} байтов.

Предположим, что сегмент находится в памяти, смещение попало в нужный интервал, тогда система Pentium прибавляет 32-разрядное поле *Base* (база) в дескрипторе к смещению, формируя то, что называется **линейным адресом**, как показано на рис. 4.42. Поле *Base* разбито на три части, которые разбросаны по дескриптору для совместимости с процессором Intel 80286, в котором поле *Base* имеет только 24 бита. В сущности, поле *Base* позволяет каждому сегменту начинаться в произвольном месте внутри 32-разрядного линейного адресного пространства.

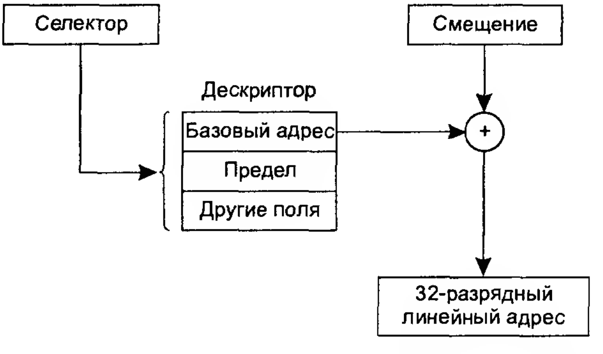


Рис. 4.42. Преобразование пары (селектор, смещение) в физический адрес

Если разбиение на страницы блокировано (с помощью бита в глобальном управляющем регистре), линейный адрес интерпретируется как физический адрес

и посылается в память для чтения или записи. Таким образом, при отключенной страничной схеме памяти мы получаем чистую схему сегментации с базовым адресом каждого сегмента, выдаваемым его дескриптором. Сегментам разрешено перекрываться случайным образом, возможно, потому, что контроль за тем, чтобы они не пересекались, мог бы причинить достаточно хлопот и занял бы слишком много времени.

С другой стороны, если доступна страничная подкачка, линейный адрес интерпретируется как виртуальный адрес и отображается на физический адрес с помощью таблицы страниц практически так же, как в наших предыдущих примерах. Единственная серьезная трудность заключается в том, что при 32-разряном виртуальном адресе и странице размером 4 Кбайт сегмент может содержать 1 млн страниц, поэтому используется двухуровневое отображение с целью уменьшения размера таблицы страниц для маленьких сегментов.

У каждой работающей программы есть **страничный каталог**, состоящий из 1024 32-разрядных записей. Он расположен по адресу, хранящемуся в глобальном регистре. Каждая запись в каталоге ссылается на таблицу страниц, также содержащую 1024 32-разрядных записей. Записи в таблицах страниц в свою очередь указывают на страничные блоки. Эта схема продемонстрирована на рис. 4.43.

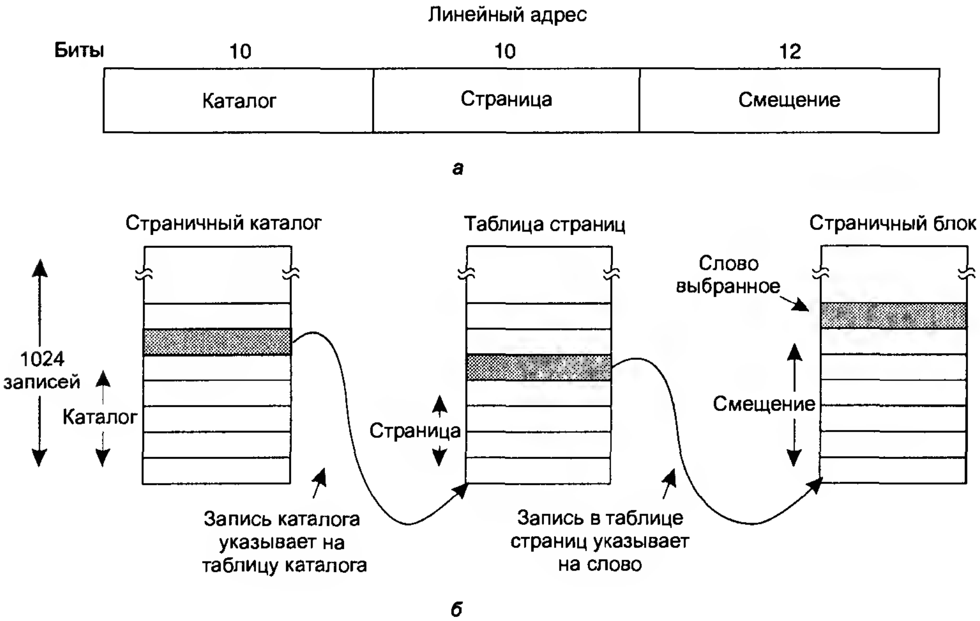


Рис. 4.43. Отображение линейного адреса на физический адрес

На рис. 4.43, *а* мы видим линейный адрес, разделенный на три поля: *Каталог*, *Страница* и *Смещение*. Поле *Каталог* используется как индекс в страничном каталоге, определяющий расположение указателя на правильную таблицу страниц. Затем поле *Страница* используется в качестве индекса в таблице страниц, чтобы найти физический адрес страничного блока. И наконец, чтобы получить

физический адрес требуемого байта или слова, к адресу страничного блока прибавляется последнее поле *Смещение*.

Каждая запись в таблице имеет размер 32 бита, 20 из которых содержат номер страничного блока. В остальные биты входят биты доступа и «грязный» бит, задаваемые аппаратурой для операционной системы, биты защиты и другие полезные биты.

Каждая таблица страниц включает в себя записи для 1024 страничных блоков размером по 4 Кбайт, таким образом, одна таблица страниц управляет четырьмя мегабайтами памяти. Сегмент, длина которого меньше 4 Мбайт, будет иметь страничный каталог с единственной записью — указателем на его единственную таблицу страниц. Следовательно, в случае короткого сегмента на поддержку таблиц страниц расходуется только две страницы вместо миллиона, который был бы нужен в одноуровневой таблице страниц.

Чтобы избежать создания повторных обращений к памяти, система Pentium, как и система MULTICS, имеет небольшой буфер быстрого преобразования адреса (TLB), который напрямую отображает наиболее часто использующиеся комбинации *Каталог-Страница* на физический адрес страничного блока. Только когда текущая комбинация отсутствует в буфере TLB, действительно выполняется механизм, показанный на рис. 4.43, и буфер TLB обновляется. Система обладает хорошей производительностью до тех пор, пока обращения к отсутствующим страницам в буфере TLB происходят относительно редко.

Также следует отметить, что если некоторые приложения не требуют сегментации, а довольствуются единым, разбитым на страницы 32-разрядным адресным пространством, эта модель все равно работает. Все сегментные регистры могут быть настроены тем же самым селектором, в дескрипторе которого поле *Base* = 0 и поле *Limit* установлено на максимум. Тогда, при использовании единственного адресного пространства, смещение команды будет линейным адресом — в сущности, обычная страничная организация памяти. Фактически все современные операционные системы для компьютера Pentium работают таким образом. Система OS/2 была единственной, которая использовала всю мощь архитектуры диспетчера памяти (MMU) фирмы Intel.

В конце концов, кто-то должен похвалить разработчиков системы Pentium. При поставленных перед ними противоречивых задачах — реализовать чистую страничную организацию памяти, чистое сегментирование и страничные сегменты и в то же время обеспечить совместимость с 286-м процессором, а кроме того, сделать все это эффективно, — результирующая структура удивительно проста и понятна.

Мы хотя и кратко, но целиком описали полную архитектуру виртуальной памяти системы Pentium и теперь следует сказать несколько слов о защите, так как эта тема тесно связана с виртуальной памятью. Как схема виртуальной памяти, так и система защиты на Pentium близка по модели к системе MULTICS. Система Pentium поддерживает четыре уровня защиты, где уровень 0 является наиболее привилегированным, а уровень 3 — наименее привилегированным. Они показаны на рис. 4.44. В каждый момент времени работающая программа находится на определенном уровне, что отмечается 2-битовым полем в его регистре слова состояния программы (PSW). Каждый сегмент в системе также имеет свой уровень.

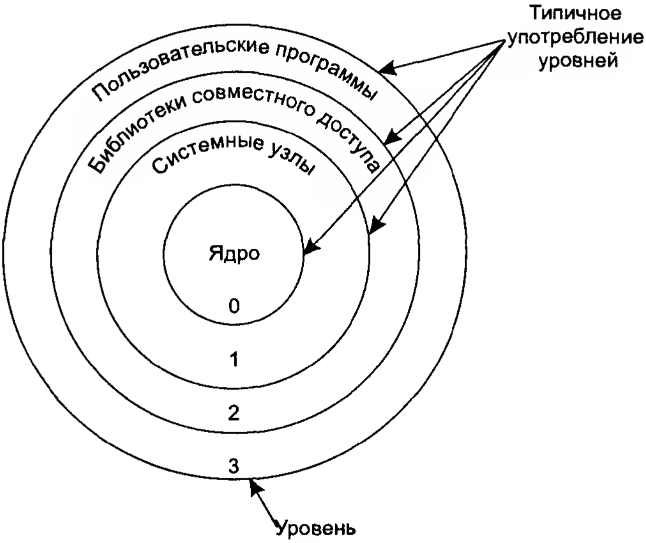


Рис. 4.44. Защита в системе Pentium

До тех пор пока программа сама ограничивает использование сегментов на своем собственном уровне, система прекрасно работает. Разрешаются попытки получения доступа к данным высшего уровня. Попытки доступа к данным более низкого уровня запрещены и вызывают прерывания. Попытки вызвать процедуры различного уровня (более высокого или низкого) позволяются, но тщательно контролируемым образом. Чтобы сделать межуровневый вызов, инструкция CALL должна содержать селектор вместо адреса. Этот селектор определяет дескриптор, называемый **шлюзом вызова** (call gate), который передает адрес вызываемой процедуры. Таким образом, перепрыгнуть в середину произвольного сегмента кода другого уровня невозможно. Могут использоваться только официальные точки входа. Концепция уровней защиты и схем вызова впервые появилась в системе MULTICS, где они представали в виде **колец защиты**.

Типичное использование этого механизма представлено на рис. 4.44. На уровне 0 мы находим ядро операционной системы, занимающееся обработкой операций ввода-вывода, управлением памятью и другими наиболее важными вопросами. На уровне 1 находится обработчик системных вызовов. На этом уровне пользовательские программы могут обращаться к процедурам для выполнения системных вызовов, но только к определенному и защищенному списку процедур. Уровень 2 содержит библиотечные процедуры, возможно, совместно используемые несколькими запущенными программами. Пользовательские программы могут вызывать эти процедуры и читать их данные, но не могут их изменять. И наконец, пользовательские программы работают на уровне 3, который имеет наименьшую степень защиты.

Эмулированные и аппаратные прерывания используют механизм, аналогичный шлюзам вызовов. Они тоже обращаются к дескрипторам, а не к абсолютным адресам, эти дескрипторы указывают на определенные процедуры. Поле *Тип* на рис. 4.40 позволяет различать программные сегменты, сегменты данных и различных видов шлюзов вызовов.

Исследования в области управления памятью

Управление памятью, особенно алгоритмы страничной подкачки, когда-то было плодотворной областью для исследований, но, кажется, большая часть этих изысканий (по крайней мере, для универсальных систем) сейчас отмерла. Реальные системы чаще всего используют некоторые разновидности алгоритма «часы», потому что он прост в реализации и относительно эффективен. Есть, правда, одно свежее исключение — доработка системы виртуальной памяти 4.4 BSD [79].

Тем не менее до сих пор проводятся исследования, касающиеся страничной организации памяти в универсальных и новейших видах систем. Некоторые из этих работ направлены на то, чтобы позволить пользовательским процессам обрабатывать свои собственные страничные прерывания и осуществлять свое собственное управление памятью, возможно, некоторым специальным способом [110]. Одной из областей, где могут понадобиться приложения, осуществляющие собственную организацию страничной структуры, являются мультимедийные системы, поэтому некоторые исследования на эту тему рассматривались в [142]. Другой областью, которая имеет некоторые особенные требования, являются портативные персональные средства связи ([3, 354]). И последняя область — это системы с 64-разрядным адресным пространством, разделяемым множеством процессов [321].

Резюме

В этой главе было проанализировано управление памятью. Мы увидели, что в простейших системах вообще нет свопинга или страничной организации памяти. Программа, загруженная в память, остается там до своего завершения. Некоторые операционные системы позволяют находиться в памяти одновременно только одному процессу, в то время как другие поддерживают многозадачность.

Следующим шагом является свопинг (то есть загрузка и выгрузка целых процессов). Когда используется подкачка такого вида, система может обрабатывать большее количество процессов, чем то, для которого достаточно пространства в памяти. Процессы, для которых нет места в памяти, целиком выгружаются на диск. Свободные области в памяти и на диске могут отслеживаться с помощью битового массива или списка свободных участков.

Современные компьютеры часто поддерживают некоторую форму виртуальной памяти. В простейшем виде адресное пространство каждого процесса делится на части постоянного размера, называемые страницами, которые могут размещаться в любом доступном страничном блоке в памяти. Существует множество алгоритмов замещения страниц, два наилучших — это алгоритмы «старения» и WSClock.

Страничные системы можно смоделировать отделением последовательности страничных обращений от программы и использованием той же самой строки обращений в различных алгоритмах. Эти модели могут использоваться для некоторых прогнозов о характеристиках страничной подкачки.

Для хорошей работы страничных систем недостаточно выбора алгоритма; кроме этого необходимо обратить внимание на такие вопросы, как определение рабочего набора, стратегию предоставления памяти и размер страниц.

Сегментация помогает в управлении структурами данных, изменяющими свой размер во время выполнения, и упрощает процессы компоновки и совместного доступа. Она также облегчает предоставление различных видов защиты разным сегментам. Иногда сегментация и разбивка на страницы комбинируются, чтобы обеспечить двумерную виртуальную память. Системы MULTICS и Intel Pentium поддерживают сегментацию и страничную организацию памяти.

Вопросы

1. Компьютерная система имеет достаточно места для того, чтобы содержать в оперативной памяти четыре программы. Эти программы простаивают в ожидании ввода-вывода половину времени. Какая часть времени работы центрального процессора пропадает?
2. На рис. 4.20 мы видим пример того, как несколько задач могут работать параллельно и при этом закончиться быстрее, чем если бы они были запущены последовательно. Предположим, что одновременно запускаются два задания, каждому из которых нужно 10 мин работы процессора. Сколько времени потребуется для завершения их работы, если они работают последовательно? А сколько, если они работают параллельно? Предположим, ожидание ввода-вывода составляет 50 %.
3. Системы подкачки устраняют свободные участки с помощью уплотнения. Предположим, что множество свободных участков и множество сегментов данных распределены случайно, а время для чтения 32-разрядного слова в памяти или записи туда равно 10 нс. Сколько примерно времени займет уплотнение 128 Мбайт в этом случае? Для простоты считаем, что слово 0 — это часть незанятой области и что самое старшее слово памяти содержит действительные данные.
4. В этой задаче сравните количество места, необходимого для учета свободной памяти при помощи битового массива и с помощью связного списка. Память размером 128 Мбайт предоставляется блоками по n байт. Для связного списка предполагается, что память состоит из чередующейся последовательности сегментов и свободных областей, каждая по 64 Кбайт. Также считаем, что для каждого узла в связном списке необходим 32-разрядный адрес в памяти, 16 разрядов для длины и 16 разрядов для поля ссылки на следующий узел. Сколько потребуется байтов для хранения структур в каждом случае? Какой метод лучше?
5. Рассмотрим систему обычной подкачки, в памяти которой содержатся свободные участки следующих размеров и в следующем порядке: 10 Кбайт, 4 Кбайт, 20 Кбайт, 18 Кбайт, 7 Кбайт, 9 Кбайт, 12 Кбайт и 15 Кбайт. Какой из них будет выбран для успешного удовлетворения запроса сегмента размером
   а) 12 Кбайт,
   б) 10 Кбайт,
   в) 9 Кбайт

по алгоритму «первый подходящий»? Ответьте на тот же самый вопрос для алгоритмов «самый подходящий», «самый неподходящий» и «следующий подходящий».

6. В чем разница между физическим адресом и виртуальным?
7. Для каждого из следующих десятичных виртуальных адресов вычислите номер виртуальной страницы и смещение, если размер страницы равен 4 Кбайт или 8 Кбайт: 20 000, 32 768, 60 000.
8. Используя таблицу страниц на рис. 4.10, сосчитайте физический адрес, соответствующий каждому из следующих виртуальных адресов:

   а) 20,

   б) 4100,

   в) 8300.
9. Процессор Intel 8086 не поддерживает виртуальную память. Тем не менее некоторые компании ранее продавали системы, содержащие неизмененный процессор 8086 и выполняющие страничную подкачку. Предложите гипотезу того, как они это делали. *Подсказка*: подумайте о логическом расположении диспетчера памяти (MMU).
10. Объем пространства на диске, который должен быть доступен для хранения страниц, связан с максимальным количеством процессов n, количеством байтов в виртуальном адресном пространстве v и числом байтов в оперативной памяти r. Выведите формулу требований на дисковое пространство в худшем случае. Насколько эта величина реалистична?
11. Считая, что команда выполняется за 10 нс, а страничное прерывание требует дополнительно n нс, напишите выражение для фактического времени выполнения команды с учетом того, что прерывания происходят каждые k инструкций.
12. Машина имеет 32-разрядное адресное пространство и страницы размером 8 Кбайт. Таблица страниц целиком поддерживается аппаратно, на запись в ней отводится одно 32-разрядное слово. При запуске процесса таблица страниц копируется из памяти в аппаратуру, одно слово требует 100 нс. Если каждый процесс работает в течение 100 мс (включая время загрузки таблицы страниц), какая доля времени процессора жертвуется на загрузку таблицы страниц?
13. Компьютер с 32-разрядным адресом использует двухуровневую таблицу страниц. Виртуальные адреса расщепляются на 9-разрядное поле верхнего уровня таблицы, 11-разрядное поле второго уровня таблицы страниц и смещение. Чему равен размер страниц и сколько их в адресном пространстве?
14. Предположим, что 32-разрядный виртуальный адрес разбивается на четыре поля: a, b, c и d. Первые три используются для трехуровневой системы таблиц страниц. Четвертое поле — это смещение. Зависит ли количество страниц от размера всех четырех полей? Если нет, то какие из полей имеют значение, а какие нет?

15. Компьютер поддерживает 32-разрядные виртуальные адреса и страницы размером 4 Кбайт. Программа и данные вместе умещаются в самую младшую страницу (0–4095). Стек размещается в самой старшей странице. Сколько записей в таблице страниц необходимо для этого процесса, если используется традиционная (одноуровневая) страничная структура? Сколько записей в таблице страниц требуется при двухуровневой страничной структуре, где каждая часть — 10-разрядная?

16. Ниже показан график выполнения фрагмента программы для компьютера с размером страницы 512 байт. Программа расположена по адресу 1020, указатель стека равен 8192 (стек увеличивается по направлению к нулю). Напишите последовательность страничных обращений, создаваемую этой программой. Каждая инструкция занимает 4 байта (1 слово), включая непосредственные константы. В последовательности обращений учитываются обращения как к инструкциям, так и к данным.

    1. Загрузить слово 6144 в регистр 0.
    2. Поместить содержимое регистра 0 в стек.
    3. Вызвать процедуру по адресу 5120, помещая в стек адрес возврата.
    4. Вычесть константу 16 из указателя стека.
    5. Сравнить полученный результат с константой 4.
    6. При равенстве перейти на адрес 5152.

17. Компьютер, чьи процессы имеют 1024 страницы в своем адресном пространстве, хранит таблицы страниц в памяти. На чтение слова из таблицы страниц требуется 5 нс. Чтобы уменьшить затраты, в компьютере существует буфер быстрого преобразования адреса (TLB), содержащий 32 пары (виртуальная страница, физический страничный блок), который может выполнить поиск за 1 нс. При какой частоте обращений к памяти, успешно реализуемых в TLB, средние затраты будут ниже 2 нс?

18. Буфер быстрого преобразования адреса на машинах VAX не содержит бита *R*. Почему?

19. Как может устройство ассоциативной памяти, требующееся для буфера TLB, быть реализовано аппаратно и каковы последствия такой конструкции для расширяемости?

20. Машина поддерживает 48-разрядные виртуальные адреса и 32-разрядные физические адреса. Размер страницы равен 8 Кбайт. Сколько требуется записей в таблице страниц?

21. Компьютер с размером страницы 8 Кбайт, размером оперативной памяти 256 Кбайт и размером виртуального адресного пространства 64 Гбайт использует инвертированную таблицу страниц для реализации своей виртуальной памяти. Насколько большой должна быть хэш-таблица, чтобы обеспечить среднее значение длины хэш-цепочки меньше 1? Предположим, что размер хэш-таблицы является степенью двойки.

22. Студент курса конструирования компиляторов предложил профессору проект написания компилятора, получающего список страничных обращений,

который может использоваться для реализации оптимального алгоритма замещения страниц. Это возможно? Почему «да» или почему «нет»? Существует ли какой-нибудь способ, который мог бы повысить эффективность страничной подкачки во время работы?

23. Если используется алгоритм замещения страниц FIFO в системе с четырьмя страничными блоками и восемью страницами, сколько страничных прерываний произойдет для последовательности обращений 0172327103 при условии, что четыре страничных блока изначально пусты? Теперь решите эту задачу для алгоритма LRU.
24. Рассмотрим последовательность страниц на рис. 4.15, *б*. Предположим, что биты R для страниц от B до A соответственно равны 11011011. Какая страница будет следующим кандидатом на удаление?
25. У маленького компьютера четыре страничных блока. Во время первого тика часов биты R равны 0111 (у страницы 0 бит R равен 0, у остальных — 1). Во время последующих тиков часов биты R принимают значения 1011, 1010, 1101, 0010, 1010, 1100 и 0001. Считая, что используется алгоритм старения с 8-разрядным счетчиком, напишите четыре значения, которые примет счетчик после последнего тика.
26. Предположим, что на рис. 4.20 $\tau = 400$. Какая страница будет удалена?
27. В алгоритме WSClock на рис. 4.21, *в* стрелка указывает на страницу с битом $R = 0$. Если $\tau = 400$, будет удалена эта страница? А если $\tau = 1000$?
28. Сколько времени займет загрузка с диска программы размером 64 Кбайт, если его среднее время поиска равно 10 мс, время вращения — 10 мс, каждая дорожка содержит 32 Кбайт

    а) для размера страницы 2 Кбайт?

    б) для размера страницы 4 Кбайт?

    Страницы раскиданы по диску случайно, и количество цилиндров так велико, что можно игнорировать вариант, при котором две страницы оказываются на одном и том же цилиндре.
29. Компьютер имеет четыре страничных блока. Время загрузки, время последнего доступа и биты R и M для каждой страницы показаны ниже (время считается в тиках системных часов):

Страница	Загружена	Последнее обращение	R	M
0	126	280	1	0
1	230	265	0	01
2	140	270	0	0
3	110	285	1	1

    а) Какую страницу выгрузит алгоритм NRU?

    б) Какую страницу выгрузит алгоритм FIFO?

    в) Какую страницу выгрузит алгоритм LRU?

    г) Какую страницу выгрузит алгоритм «вторая попытка»?

30. Одна из первых машин с системой разделения времени PDP-1 имела память 4 К 18-разрядных слов. В каждый конкретный момент времени она содержала в памяти один процесс. Когда планировщик решал запустить другой процесс, процесс в памяти записывался на страничный барабан с 4 К 18-разрядных слов по окружности барабана. Барабан мог начать запись (или чтение) с любого слова, а не только со слова 0. Как вы полагаете, почему был выбран этот барабан?
31. Компьютер обеспечивает каждый процесс 65 536 байт адресного пространства, разделенного на страницы по 4096 байт. Некая программа имеет размер текста 32 768 байт, размер данных 16 386 байт и размер стека 15 870 байт. Поместится ли эта программа в адресном пространстве? А если бы размер страницы был 512 байт, она поместилась бы? Помните, что страница не может вмещать части двух разных сегментов.
32. Может ли страница оказаться в двух рабочих наборах одновременно? Объясните.
33. Если страница совместно используется двумя процессами, может ли она быть страницей «только для чтения» для одного процесса и «чтение-запись» для другого процесса? Почему да (или нет)?
34. Было замечено, что количество инструкций, выполненных между страничными прерываниями, прямо пропорционально количеству страничных блоков, предоставленных программе. Если доступная память увеличивается вдвое, то средний интервал между страничными прерываниями также увеличивается вдвое. Предположим, что нормальная инструкция занимает 1 мкс, но если происходит страничное прерывание, она выполняется за 2001 мкс (то есть 2 мс идут на обработку прерывания). Если программа требует для работы 60 с, во время которой она вызывает 15 000 страничных прерываний, сколько времени она заняла бы, если бы было доступно удвоенное количество исходной памяти?
35. Группа разработчиков операционных систем для компании Frugal Computer Company размышляют о способе уменьшения количества резервного пространства для хранения, необходимого в их операционной системе. Ведущий специалист предложил вообще не беспокоиться о сохранении текста программы в области подкачки, а просто загружать его страницами прямиком из двоичного файла всякий раз, когда он требуется. При каком условии, если оно существует, эта идея работает для текста программы? А при каком условии, опять же, если оно существует, она работает для данных?
36. Инструкция машинного языка, загружающая 32-разрядное слово в регистр, содержит 32-разрядный адрес этого слова. Какое максимальное количество страничных прерываний может вызвать данная команда?
37. Объясните разницу между внутренней и внешней фрагментацией. Какая из них происходит в страничных системах? А какая имеет место в системах, использующих чистую сегментацию?
38. Когда поддерживаются и сегментация, и страничная организация памяти, как в системе MULTICS, сначала должен быть найден дескриптор сегмента,

затем идентификатор страницы. Может ли при таком двухуровневом поиске работать также буфер быстрого преобразования адреса (TLB)?

39. Составьте гистограмму и вычислите средний и медиановый размеры выполнимых бинарных файлов на своем компьютере. В системе Windows учтите все файлы с расширениями .exe и .dll, в системе UNIX рассмотрите все выполняемые файлы в каталогах */bin*, */usr/bin* и */local/bin*, которые не являются сценариями (или используйте утилиту *file*, чтобы найти все выполняемые файлы). Определите оптимальный размер страницы для этого компьютера, принимая во внимание только код (не данные). Рассмотрите внутреннюю фрагментацию и размер таблицы страниц, сделав некоторые разумные предположения о размере записи в таблице страниц. Считайте, что все программы запускаются с равной вероятностью и таким образом должны учитываться в расчетах с равным весом.

40. Маленькие программы для системы MS-DOS могут быть скомпилированы как *.COM*-файлы. Такие файлы всегда загружаются по адресу 0x100 в единственный сегмент памяти, который используется для кода, данных и стека. У инструкций передачи управления, таких как JMP и CALL, или обращающихся к статическим данным по фиксированным адресам, адреса скомпилированы в объектный код. Напишите программу, которая может перенастроить адреса в таком программном файле, чтобы выполнить его запуск с произвольного адреса. Ваша программа должна просматривать текст нужной программы, ища объектные коды команд, обращающихся к фиксированным адресам памяти, затем изменять те адреса, которые указывают на места в памяти внутри перемещаемого диапазона. Вы можете найти объектные коды в тексте программы на ассемблере. Заметим, что в общем случае выполнение этой задачи полностью и без дополнительной информации невозможно, потому что некоторые слова данных могут иметь значения, совпадающие с кодами объектных инструкций.

41. Напишите программу, моделирующую страничную систему. При запуске программы следует спросить пользователя об алгоритме страничного замещения, выбирая из FIFO, LRU и, по крайней мере, еще одного. На каждом цикле считывайте номер страницы, к которой обращаются, из файла. Сформируйте листинг, аналогичный рис. 4.23, но повернутый на 90°, так чтобы каждая новая страница увеличивала длину выходного файла на одну строку.

42. Напишите программу, моделирующую алгоритм последовательности расстояний, описанный в тексте. Входные данные программы — список страничных обращений (содержащихся в файле) и номер страничного блока в доступной физической памяти. Если возможно, используйте последовательность обращений к данным из действительной программы вместо сформированных случайно страничных обращений. Программа должна содержать стек страниц, аналогичный рис. 4.23. При каждом страничном прерывании для выбора замещаемой страницы должна вызываться процедура. По завершении работы программа должна печатать последовательность расстояний аналогично рис. 4.24. Запустите вашу программу несколько раз для различных размеров памяти и посмотрите, какие выводы вы можете сделать.

Глава 5
Ввод-вывод

Одна из важнейших функций операционной системы состоит в управлении всеми устройствами ввода-вывода компьютера. Операционная система должна давать этим устройствам команды, перехватывать прерывания и обрабатывать ошибки. Она должна также обеспечить простой и удобный интерфейс между устройствами и остальной частью системы. Интерфейс, насколько это возможно, должен быть одинаковым для всех устройств (для достижения независимости от применяемых устройств). Программное обеспечение ввода-вывода составляет существенную часть операционной системы. Тому, как операционная система управляет устройствами ввода-вывода, и посвящена эта глава.

Глава организована следующим образом. Сначала мы рассмотрим некоторые основы аппаратуры ввода-вывода, затем в общих чертах познакомимся с программным обеспечением ввода-вывода. Программное обеспечение ввода-вывода может быть структурировано в виде уровней, у каждого из которых есть строго очерченный круг задач. Мы рассмотрим эти уровни, чтобы понять, что они делают и как согласуются друг с другом.

После этого вступления мы подробно ознакомимся с некоторыми устройствами ввода-вывода: дисками, часами, клавиатурами и дисплеями. Для каждого устройства мы рассмотрим его аппаратную часть и программное обеспечение. Затем мы затронем вопрос управления питанием.

Принципы аппаратуры ввода-вывода

Разные специалисты рассматривают аппаратуру ввода-вывода с различных точек зрения. Инженеры-электронщики видят в них микросхемы, провода, источники питания, двигатели и прочие физические компоненты, из которых и состоит аппаратура. Программисты в первую очередь обращают внимание на интерфейс, предоставляемый программному обеспечению, — команды, принимаемые аппаратурой, выполняемые ею функции и ошибки, о которых аппаратура может сообщить. В этой книге нас интересует именно программирование устройств ввода-вывода, а не их проектирование, построение или поддержка. Поэтому сфера наших интересов будет ограничена тем, как программировать аппаратуру, и в нее не входят физические принципы работы аппаратуры. В то же время программирование многих устройств ввода-вывода часто оказывается тесно связанным с их внутренним функционированием. В следующих трех разделах мы кратко предоставим общие

основы знаний из области аппаратуры ввода-вывода, касающиеся программирования. Этот материал можно рассматривать в качестве обзорного и продолжения темы раздела «Обзор аппаратного обеспечения компьютера» главы 1.

Устройства ввода-вывода

Устройства ввода-вывода можно грубо разделить на две категории: **блочные устройства** и **символьные устройства**. Блочными называются устройства, хранящие информацию в виде блоков фиксированного размера, причем у каждого блока имеется адрес. Обычно размеры блоков варьируются от 521 до 32 768 байт. Важное свойство блочного устройства состоит в том, что каждый его блок может быть прочитан независимо от остальных блоков. Наиболее распространенными блочными устройствами являются диски.

Если приглядеться внимательнее, то окажется, что граница между блок-адресуемыми устройствами и устройствами, к отдельным блокам которых нельзя адресоваться напрямую, не определена строго. Все согласны с тем, что диск является блок-адресуемым устройством, так как вне зависимости от текущего положения головки дисковода всегда можно переместить ее на определенный цилиндр и затем считать или записать отдельный блок с нужной дорожки. Рассмотрим теперь накопитель на магнитной ленте (магнитофон), применяемый для хранения резервных копий диска. На ленте хранится последовательность блоков. Если магнитофону дать команду прочитать блок *N*, он всегда может перемотать ленту и начать читать блоки, пока не дойдет до запрашиваемого блока *N*. Эта операция подобна поиску блока на диске с той лишь разницей, что она занимает значительно больше времени. Кроме того, в зависимости от накопителя и формата хранящихся на нем данных, может оказаться возможной или невозможной запись отдельного произвольного блока в середине ленты. Даже если и было бы возможно использовать магнитные ленты в качестве блочных устройств произвольного доступа, это являлось бы в какой-то степени натяжкой: никто их не использует таким образом.

Другой тип устройств ввода-вывода — символьные устройства. Символьное устройство принимает или предоставляет поток символов без какой-либо блочной структуры. Оно не является адресуемым и не выполняет операцию поиска. Принтеры, сетевые интерфейсные карты, мыши (для указания точки на экране), крысы (для лабораторных экспериментов по психологии) и большинство других устройств, не похожих на диски, можно рассматривать как символьные устройства.

Такая схема классификации не совершенна. Некоторые устройства просто не попадают ни в одну из категорий. Например, часы не являются блок-адресуемыми. Они также не формируют и не принимают символьных потоков. Вся их работа состоит в инициировании прерываний в строго определенные моменты времени. Экраны отображения памяти также не втискиваются в рамки этой модели. И все же модель блочных и символьных устройств является настолько общей, что может использоваться в качестве основы для достижения независимости от устройств некоторого программного обеспечения операционных систем, имеющего дело с вводом-выводом. Например, файловая система имеет дело с абстрактными блочными устройствами, а зависимую от устройств часть составляет программному обеспечению низкого уровня.

Устройства ввода-вывода покрывают огромный диапазон скоростей, что создает определенные трудности для программного обеспечения, которому приходится обеспечивать хорошую производительность на скоростях передачи данных, различающихся несколькими порядками. В табл. 5.1 приведены скорости данных для некоторых часто встречающихся устройств. Со временем у многих устройств появляются все более быстрые новые модели.

Таблица 5.1. Скорости данных типичных устройств

Устройство	Скорость данных
Клавиатура	10 байт/с
Мышь	100 байт/с
Модем 56 К	7 Кбайт/с
Телефонная линия	8 Кбайт/с
Двойная линия ISDN	16 Кбайт/с
Лазерный принтер	100 Кбайт/с
Сканер	400 Кбайт/с
Классическая сеть Ethernet	1,25 Мбайт/с
Шина USB (Universal Serial Bus)	1,5 Мбайт/с
Цифровая видеокамера	4 Мбайт/с
IDE-диск	5 Мбайт/с
40x CD-ROM	6 Мбайт/с
Быстрая сеть Ethernet	12,5 Мбайт/с
Шина ISA	16,7 Мбайт/с
IDE-диск (ATA-2)	16,7 Мбайт/с
FireWire (IEEE 1394)	50 Мбайт/с
XGA-монитор	60 Мбайт/с
Сеть SONET OC-12	78 Мбайт/с
Диск SCSI Ultra 2	80 Мбайт/с
Гигабитная сеть Ethernet	125 Мбайт/с
Лента Ultrium	320 Мбайт/с
Шина PCI	528 Мбайт/с
Объединительная плата Sun Gigaplane XB	20 Гбайт/с

Контроллеры устройств

Устройства ввода-вывода обычно состоят из механической части и электронной части. Часто эти части можно разделить для придания модели более модульного и общего вида. Электронный компонент устройства называется **контроллером устройства** или **адаптером**. В персональных компьютерах он часто принимает форму печатной платы, вставляемой в слот расширения. Механический компонент находится в самом устройстве. Такая организация показана на рис. 1.5.

Плата контроллера обычно снабжается разъемом, к которому может быть подключен кабель, ведущий к самому устройству. Многие контроллеры способны управлять двумя, четырьмя или даже восемью идентичными устройствами. Если

интерфейс между контроллером и устройством является стандартным, то есть официальным стандартом ANSI, IEEE или ISO либо фактическим стандартом, тогда различные компании могут выпускать отдельно контроллеры и устройства, удовлетворяющие данному интерфейсу. Так, многие компании производят жесткие диски, соответствующие интерфейсу IDE или SCSI.

Интерфейс между устройством и контроллером часто является интерфейсом очень низкого уровня. Например, какой-нибудь жесткий диск может быть отформатирован по 256 секторов на дорожку, с размером секторов по 512 байт. В действительности с диска в контроллер поступает последовательный поток битов, начинающийся с **заголовка сектора** (преамбулы), за которым следует 4096 бит в секторе, и, наконец, контрольная сумма, также называемая **кодом исправления ошибок** (ECC, Error-Correcting Code). Заголовок сектора записывается на диск во время форматирования. Он содержит номера цилиндра и сектора, размер сектора, информацию синхронизации и т. п.

Работа контроллера заключается в конвертировании последовательного потока битов в блок байтов и выполнение коррекции ошибок, если это необходимо. Обычно байтовый блок собирается бит за битом в буфере контроллера. Затем проверяется контрольная сумма блока, и если она совпадает с указанной в заголовке сектора, блок объявляется считанным без ошибок, после чего он копируется в оперативную память.

Контроллер монитора (видеоадаптер) также работает как бит-последовательное устройство, на таком же низком уровне. Он считывает в памяти байты, содержащие символы, которые следует отобразить, и формирует сигналы, используемые для модуляции луча электронной трубки, заставляющие ее выводить изображение на экран. Видеоадаптер также формирует сигналы, управляющие горизонтальным и вертикальным возвратом электронного луча. Если бы ни контроллер, программисту пришлось бы управлять перемещениями аналогового электронного луча. В действительности же операционная система всего лишь инициализирует контроллер, задавая небольшое число параметров, таких как количество символов или пикселов в строке и число строк на экране, а всю тяжелую работу по управлению передвижениями электронного луча по экрану выполняет контроллер.

Отображаемый на адресное пространство памяти ввод-вывод

У каждого контроллера есть несколько регистров, с помощью которых с ним может общаться центральный процессор. При помощи записи в эти регистры операционная система велит устройству предоставить данные, принять данные, включиться или выключиться и т. п. Читая из этих регистров, операционная система может узнать состояние устройства, например готово ли оно к приему новой команды и т. д.

Помимо управляющих регистров, у многих устройств есть буфер данных, из которого операционная система может читать данные, а также писать данные в него. Например, для отображения пикселов на экране данные обычно помещаются в видеопамять, являющуюся, по сути, буфером данных, доступным операционной системе и другим программам для чтения и записи.

Существует два альтернативных способа реализации доступа к управляющим регистрам и буферам данных устройств ввода-вывода. Первый вариант заключается в том, что каждому управляющему регистру назначается номер **порта ввода-вывода**, 8- или 16-разрядное целое число. При помощи такой специальной команды процессора, как

```
IN REG.PORT
```

центральный процессор может прочитать управляющий регистр устройства из порта PORT в регистр процессора REG. Аналогично с помощью команды

```
OUT PORT.REG
```

центральный процессор может записать содержимое своего регистра REG в управляющий регистр устройства через порт PORT. Подобным образом работали самые древние компьютеры, включая почти все мэйнфреймы, такие как IBM 360 и его преемники.

При такой схеме адресные пространства оперативной памяти и устройств ввода-вывода не пересекаются, как видно из рис. 5.1, *а*. Команды

```
IN R0.4
```

и

```
MOV R0.4
```

выполняют принципиально различные действия. Первая команда читает содержимое порта ввода-вывода 4 в регистр R0, тогда как вторая читает в этот же регистр содержимое слова памяти по адресу 4. Таким образом, четверки в этих командах означают различные адреса из непересекающихся адресных пространств.

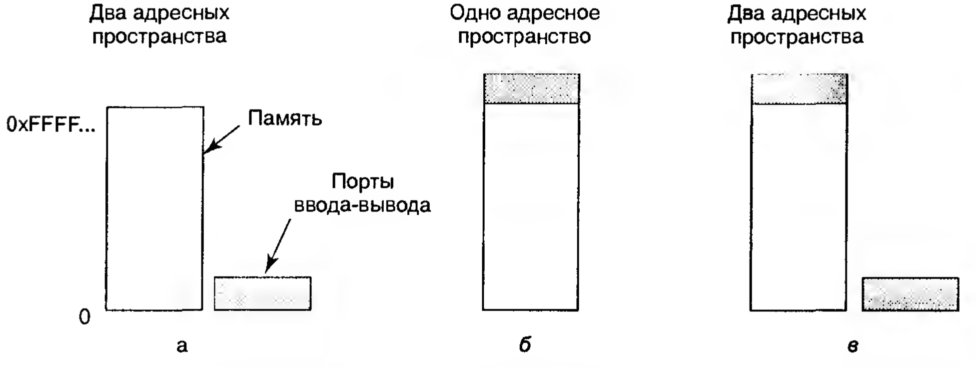


Рис. 5.1. Раздельные адресные пространства (а); отображаемый на адресное пространство памяти ввод-вывод (*б*); гибрид (*в*)

Второй подход, впервые реализованный в компьютере PDP-11, состоял в отображении всех управляющих регистров периферийных устройств на адресное пространство памяти, как показано на рис. 5.1, *б*. Каждому управляющему регистру назначался уникальный адрес в памяти. Такая система называется **отображаемым на адресное пространство памяти вводом-выводом**. Обычно для регистров устройств отводятся адреса на вершине адресного пространства. Также существу-

ют различные гибридные схемы, с отображаемыми на адресное пространство памяти буферами данных и отдельными портами ввода-вывода (рис. 5.1, *в*). Эта схема довольно широко применяется, например, в совместимых с IBM PC компьютерах на базе процессоров x86 и Pentium, в которых, помимо портов ввода-вывода с номерами от 0 до 64 К, адресное пространство оперативной памяти от 640 К до 1 М зарезервировано под буферы данных устройств ввода-вывода.

Как работают все эти схемы? Во всех случаях, когда центральный процессор хочет прочитать слово данных либо из памяти, либо из порта ввода-вывода, он выставляет нужный адрес на адресную шину, после чего выставляет сигнал READ на управляющую шину. Вторая сигнальная линия позволяет отличить обращение к памяти от обращения к порту. В зависимости от состояния этой линии шины управления на запрос процессора реагирует устройство (контроллер) ввода-вывода или память. Если пространство адресов общее (как на рис. 5.1, *б*), то каждый модуль памяти и каждое устройство ввода-вывода сравнивает выставленный на шину адрес с обслуживаемым им диапазоном адресов. Если выставленный на шину адрес попадает в этот диапазон, то соответствующе устройство реагирует на запрос процессора. Поскольку выделенные внешним устройствам адреса удаляются из памяти, память не реагирует на них и конфликта адресов не происходит.

Обе схемы обращения к контроллерам имеют свои сильные и слабые стороны. Начнем с достоинств отображаемого на адресное пространство памяти ввода-вывода. Во-первых, при такой схеме для обращения к устройствам ввода-вывода не требуются специальные команды процессора, такие как IN и OUT. В результате программу, общающуюся с таким устройством, можно написать целиком на языке C или C++, без вставок на ассемблере или обращений к подпрограммам, написанным на ассемблере, то есть без дополнительных накладных расходов.

Во-вторых, при отображении регистров ввода-вывода на память не требуется специального механизма защиты от пользовательских процессов, пытающихся обращаться к внешним устройствам. Все, что нужно сделать операционной системе, — это исключить ту часть адресного пространства, на которую отображаются управляющие регистры устройств ввода-вывода из адресного пространства пользователей. Более того, если управляющие регистры различных устройств ввода-вывода отображаются на различные страницы памяти, операционная система может предоставить доступ к различным страницам различным пользователям, таким образом предоставляя пользователям доступ к одним устройствам и запрещая доступ к другим. Для этого нужно всего лишь включить номер соответствующей страницы памяти в карту памяти нужного пользователя. В результате такая схема позволяет разместить драйверы различных устройств в различных адресных пространствах, тем самым не только уменьшая размер ядра, но и удерживая драйверы от вмешательства в дела друг друга.

В-третьих, при отображении регистров ввода-вывода на память каждая команда процессора, обращающаяся к памяти, может с тем же успехом обращаться к управляющим регистрам устройства. Например, если у процессора есть в наборе команд инструкция TEST, проверяющая содержимое некоего слова в памяти на равенство 0, она может с тем же успехом применяться и при обращении к управляющему регистру устройства ввода-вывода. Управляющий регистр, равный 0, будет означать,

например, готовность данного устройства к приему новой команды. Программа на ассемблере может выглядеть следующим образом:

```
LOOP:     TEST      PORT_4    // сравнить содержимое порта 4 с нулем
          BEQ       READY     // если он равен 0, идти на метку READY
          BRANCH    LOOP      // в противном случае продолжать опрос порта
READY:
```

Если отображения регистров ввода-вывода на память нет, управляющий регистр устройства должен быть сначала считан в регистр процессора, а уже затем сравнен с 0, что требует двух команд процессора вместо одной. Для приведенного выше цикла добавление четвертой команды может слегка снизить (все зависит от конкретных процессоров, конечно) скорость реакции драйвера на появление признака готовности устройства.

В разработке компьютеров практически у любого решения есть как положительные, так и отрицательные стороны. Отображение регистров ввода-вывода на память также обладает недостатками. Во-первых, в большинстве современных компьютеров применяется кэширование памяти. Кэширование управляющих регистров привело бы просто к катастрофе. Поясним это утверждение на уже приведенном выше примере программы, опрашивающей в цикле порт PORT_4. При первом обращении к этому порту считалось бы верное значение порта, но это значение сохранилось бы в кэше. Все последующие обращения к порту PORT_4 на следующих итерациях цикла просто читали бы значение, сохраненное в кэше, и никогда не обращались бы к реальному устройству. Таким образом, программа никогда не вышла бы из цикла ожидания готовности, так как при кэшировании регистров устройств она просто не смогла бы узнать об изменении управляющего регистра.

Чтобы не допустить такой ситуации, необходима специальная аппаратура, способная выборочно запрещать кэширование, например, в зависимости от номера страницы памяти, к которой обращается процессор. Таким образом, отображение регистров ввода-вывода на память увеличивает сложность аппаратуры и операционной системы, которой приходится управлять избирательным кэшированием.

Во-вторых, при едином адресном пространстве все модули памяти и все устройства ввода-вывода должны изучать все обращения процессора к памяти, чтобы определить, на которые им следует реагировать. Если у компьютера одна общая шина (рис 5.2, *а*), реализовать подобный просмотр всех обращений к памяти всеми устройствами несложно.

Однако в конструкции современных персональных компьютеров наблюдается тенденция в сторону использования выделенной высокоскоростной шины (рис 5.2, *б*), архитектурной особенности, кстати, уже давно применявшейся в мэйнфреймах. Эта шина предназначена для увеличения скорости обмена данными между процессором и памятью, чему в архитектуре общей шины сильно мешали медленные устройства ввода-вывода. В компьютерах на базе процессора Pentium таких внешних шин целых три (шина памяти, PCI и ISA), как было показано на рис. 1.11.

Сложность применения выделенной шины памяти на машинах с отображением регистров ввода-вывода на память состоит в том, что у устройств ввода-вывода нет способа увидеть адреса памяти, выставляемые процессором на эту шину, следовательно, они не могут реагировать на такие адреса. Поэтому, чтобы отображение регистров ввода-вывода могло работать на системах с несколькими шина-

ми, необходимы специальные меры. Один способ решения этой проблемы состоит в том, что сначала все обращения к памяти посылаются процессором по выделенной быстрой шине напрямую памяти (чтобы не снижать производительности). Если память не может ответить на эти запросы, процессор пытается сделать это еще раз по другим шинам. Такое решение работоспособно, но требует дополнительного увеличения сложности аппаратуры.

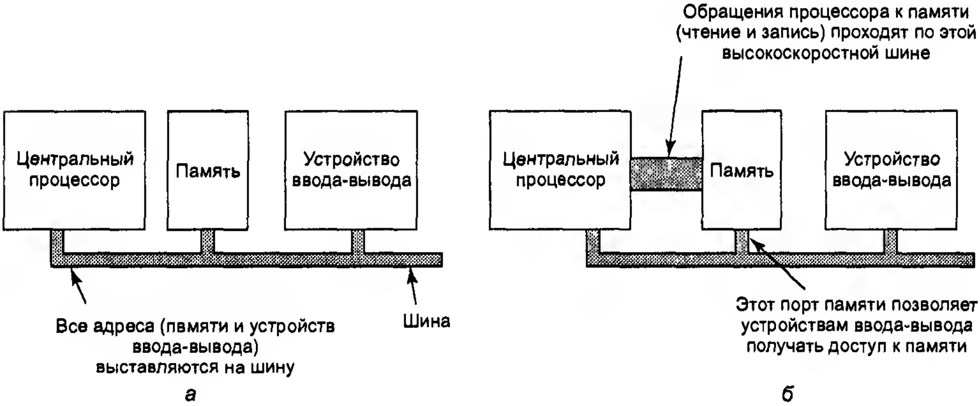


Рис. 5.2. Архитектура с одной шиной (*а*); архитектура памяти с двумя шинами (*б*)

Второе возможное решение заключается в установке на шину памяти специального следящего устройства, передающего все адреса потенциально заинтересованным устройствам ввода-вывода. Проблема, однако, в том, что устройства ввода-вывода могут просто не успеть обработать эти запросы с той же скоростью, что и память.

Третье решение, используемое в компьютерах на базе процессора Pentium (рис. 1.11), состоит в фильтрации адресов микросхемой моста PCI. Эта микросхема содержит регистры диапазона, заполняемые во время загрузки компьютера. Например, диапазон адресов от 640 К до 1 М может быть помечен как не относящийся к памяти. Все адреса, попадающие в подобный диапазон, передаются не памяти, а на шину PCI. Недостаток этой схемы состоит в необходимости принятия во время загрузки решения о том, какие адреса не являются адресами памяти. Итак, у каждой схемы есть свои достоинства и недостатки, так что компромиссы и уступки неизбежны.

Прямой доступ к памяти (DMA)

Независимо от того, отображаются ли регистры или буферы ввода-вывода на память или нет, центральному процессору необходимо как-то адресоваться к контроллерам устройств для обмена данными с ними. Центральный процессор может запрашивать данные от контроллера ввода-вывода по одному байту, но подобная организация обмена данными крайне неэффективна, так как расходует огромное количество процессорного времени. Поэтому на практике часто применяется другая схема, называемая **прямым доступом к памяти** (DMA, direct memory access).

Операционная система может воспользоваться прямым доступом к памяти только при наличии аппаратного DMA-контроллера, который есть у большинства систем. Иногда DMA-контроллер интегрируется в другие контроллеры, например в дисковый контроллер, но такой дизайн требует оснащения DMA-контроллерами каждого периферийного устройства. Как правило, DMA-контроллер, устанавливаемый на материнской плате, обслуживает запросы по передаче данных нескольких различных устройств ввода-вывода, часто на конкурентной основе.

Где бы он ни располагался физически, DMA-контроллер может получать доступ к системной шине независимо от центрального процессора, как показано на рис. 5.3. Он содержит несколько регистров, доступных центральному процессору для чтения и записи. К ним относятся регистр адреса памяти, счетчик байтов и один или более управляющих регистров. Управляющие регистры задают, какой порт ввода-вывода должен быть использован, направление переноса данных (чтение из устройства ввода-вывода или запись в него), единицу переноса (осуществлять перенос данных побайтно или пословно), а также число байтов, которые следует перенести за одну операцию.

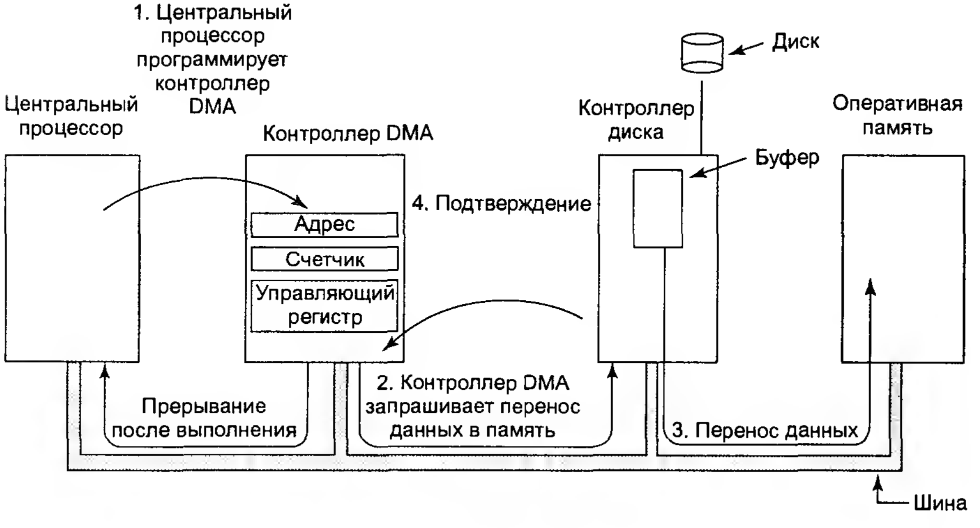


Рис. 5.3. Работа DMA-контроллера

Чтобы понять, как работает DMA, познакомимся сначала с тем, как происходит чтение с диска при отсутствии DMA. Сначала контроллер считывает с диска блок (один или несколько секторов) последовательно, бит за битом, пока весь блок не окажется во внутреннем буфере контроллера. Затем контроллер проверяет контрольную сумму, чтобы убедиться, что при чтении не произошло ошибки. После этого контроллер инициирует прерывание. Когда операционная система начинает работу, она может прочитать блок диска побайтно или пословно, в цикле сохраняя считанное слово или байт в оперативной памяти.

При использовании DMA процедура совершенно другая. Сначала центральный процессор программирует DMA-котроллер, устанавливая его регистры и указывая таким образом, какие данные и куда следует переместить (шаг *1* на рис. 5.3).

Затем процессор дает команду дисковому контроллеру прочитать данные во внутренний буфер и проверить контрольную сумму. Когда данные получены и проверены контроллером диска, DMA может начинать работу.

DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос чтения (шаг *2*). Этот запрос чтения выглядит как обычный запрос чтения, так что контроллер диска даже не знает, пришел ли он от центрального процессора или от контроллера DMA. Обычно адрес памяти уже находится на адресной шине, так что контроллер диска всегда знает, куда следует переслать следующее слово из своего внутреннего буфера. Запись в память является еще одним стандартным циклом шины (шаг *3*). Когда запись закончена, контроллер диска также по шине посылает сигнал подтверждения контроллеру DMA (шаг *4*). Затем контроллер DMA увеличивает используемый адрес памяти и уменьшает значение счетчика байтов. После этого шаги со *2*-го по *4*-й повторяются, пока значение счетчика не станет равно нулю. По завершении цикла копирования контроллер DMA инициирует прерывание процессора, сообщая ему таким образом, что перенос данных завершен. Операционной системе не нужно копировать блок диска в память. Он уже находится там.

Контроллеры DMA значительно различаются по степени своей сложности. Самые простые из них за один раз выполняют одну операцию переноса данных, как описывалось выше. Более сложные контроллеры могут выполнять сразу несколько подобных операций. У таких контроллеров несколько каналов, каждый из которых управляется своим набором внутренних регистров. Центральный процессор начинает с того, что загружает в эти регистры соответствующие параметры. Все операции переноса данных должны выполняться с различными устройствами ввода-вывода. После переноса каждого слова данных (шаги *2–4* на рис. 5.3) контроллер DMA решает, какое устройство будет им обслужено следующим. Этот выбор может производиться циклически или при помощи приоритетной схемы, предоставляющей одним устройствам преимущество по сравнению с другими. Одновременно несколько запросов могут дожидаться исполнения, при условии, что существует способ однозначно отличить подтверждения различных устройств. Часто с этой целью для каждого канала DMA используются различные линии подтверждения.

Многие шины могут работать в двух режимах: в пословном и поблочном. Некоторые контроллеры DMA также могут функционировать в обоих режимах. В пословном режиме процедура выглядит так, как описывалось выше: контроллер DMA выставляет запрос на перенос одного слова и получает его. Если центральному процессору также нужна эта шина, ему приходится подождать. Этот механизм называется **захватом цикла** (cycle stealing), потому что контроллер устройства периодически «подкрадывается» и забирает случайный цикл шины у центрального процессора, слегка его тормозя. В блочном режиме контроллер DMA велит устройству занять шину, сделать серию пересылок и отпустить шину. Такой способ действий называется **пакетным режимом**. Он более эффективен, чем захват цикла, поскольку занятие шины требует времени, а в пакетном режиме эта процедура выполняется всего один раз для передачи целого блока данных. Недостатком этого метода является то, что при переносе большого блока данных он может заблокировать центральный процессор и другие устройства на существенный промежуток времени.

В обсуждавшейся нами модели, иногда называемой **сквозным режимом**, контроллер DMA велит контроллеру устройства переслать данные напрямую в оперативную память. В некоторых DMA-контроллерах используется также режим, при котором контроллер устройства посылает слово данных контроллеру DMA, который затем выставляет на шину еще один запрос для передачи этого слова туда, куда его нужно передать. При такой схеме требуется лишний цикл шины на передачу каждого слова, зато такая схема обладает большей гибкостью, так как также позволяет выполнять копирование с устройства на устройство, минуя память, и даже из памяти в память. (Для этого нужно сначала дать команду чтения из памяти, а затем команду записи в память, но по другому адресу.)

Большинство контроллеров DMA используют для передачи данных физические адреса памяти. Чтобы использовать физические адреса памяти, операционная система должна преобразовать виртуальный адрес буфера памяти в физический и записать этот физический адрес в адресный регистр контроллера DMA. В некоторых контроллерах DMA применяется альтернативная схема, при которой в контроллер DMA записывается сразу виртуальный адрес. В этом случае контроллер DMA должен использовать менеджер памяти MMU для преобразования адреса. Виртуальный адрес может быть выставлен на адресную шину только в том случае, когда MMU является частью памяти (что возможно, но редко), а не частью центрального процессора.

Как мы уже упоминали, до начала операции DMA диск сначала считывает данные в свой внутренний буфер. Возможно, вы задаетесь вопросом, почему контроллер не помещает данные прямо в оперативную память, по мере получения их с диска. Другими словами, зачем ему нужен внутренний буфер? Тому есть две причины. Во-первых, при помощи внутренней буферизации контроллер диска может проверить контрольную сумму до начала переноса данных в память. Если контрольные суммы не совпадают, формируется сигнал об ошибке и перенос данных не производится.

Во-вторых, дело в том, что как только началась операция чтения с диска, биты начинают поступать с постоянной скоростью, независимо от того, готов контроллер диска их принимать или нет. Если контроллер диска попытается писать эти данные напрямую в память, ему придется делать это по системной шине. Если при передаче очередного слова шина окажется занятой каким-либо другим устройством (например, использующим ее в пакетном режиме), контроллеру диска придется ждать. Если следующее слово с диска прибудет раньше, чем контроллер успеет сохранить предыдущее, контроллер либо потеряет предыдущее слово, либо ему придется сохранять его где-либо еще. Таким образом, необходимость внутреннего буферирования становится очевидной. При наличии внутреннего буфера контроллеру диска шина не нужна до тех пор, пока не начнется операция DMA. В результате устройство контроллера диска оказывается проще, так как при операции DMA пересылки данных параметр времени не является критичным. (Некоторые древние контроллеры действительно напрямую обращались к памяти, обладая внутренним буфером небольшого размера, что часто приводило к ошибкам перегрузки при занятости шины.)

DMA используется не во всех компьютерах. Главный аргумент против использования DMA состоит в том, что центральный процессор обычно значительно

превосходит DMA-контроллер по скорости и может выполнить ту же работу значительно быстрее (если только скорость ограничена не быстродействием устройства ввода-вывода). При отсутствии другой работы у центрального процессора заставлять быстрый центральный процессор ждать, пока медленный контроллер DMA выполнит свою работу, бессмысленно. Кроме того, компьютер без контроллера DMA, с центральным процессором, выполняющим всю работу программно, оказывается дешевле, что крайне важно в производстве компьютеров нижней ценовой категории (например, встроенных).

Еще раз о прерываниях

Мы кратко упомянули прерывания в разделе «Устройства ввода-вывода» главы 1, но о них следует сказать еще несколько слов. Структура прерываний типичной персональной компьютерной системы проиллюстрирована на рис. 5.4. На аппаратном уровне прерывания работают следующим образом. Когда устройство ввода-вывода заканчивает свою работу, оно инициирует прерывание (при условии, что прерывания разрешены операционной системой). Для этого устройство выставляет сигнал на выделенную устройству специальную линию шины. Этот сигнал распознается микросхемой контроллера прерываний, расположенной на материнской плате. Контроллер прерываний принимает решение о дальнейших действиях.

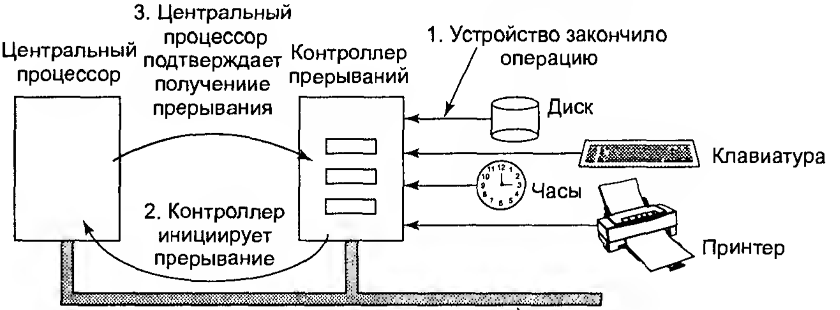


Рис. 5.4. Схема прерываний в компьютере. Соединения между устройствами и контроллером прерываний в действительности являются специальными линиями шины, а не выделенными проводами

При отсутствии других необработанных запросов прерывания контроллер прерываний обрабатывает прерывание немедленно. Если прерывание уже обрабатывается, и в это время приходит запрос от другого устройства по линии с более низким приоритетом, то новый запрос просто игнорируется. В этом случае устройство продолжает удерживать сигнал прерывания на шине до тех пор, пока оно не будет обслужено центральным процессором.

Для обработки прерывания контроллер выставляет на адресную шину номер устройства, требующего к себе внимания, и устанавливает сигнал прерывания на соответствующий контакт процессора.

Этот сигнал заставляет процессор приостановить текущую работу и начать выполнять обработку прерывания. Номер, выставленный на адресную шину,

используется в качестве индекса в таблице, называемой **вектором прерываний**, из которой извлекается новое значение счетчика команд. Новый счетчик команд указывает на начало соответствующей процедуры обработки прерывания. Обычно с этого места аппаратные и эмулированные прерывания используют один и тот же механизм и часто пользуются одним и тем же вектором. Расположение вектора может быть либо жестко прошито на аппаратном уровне, либо, наоборот, располагаться в произвольном месте памяти, на которое указывает специальный регистр процессора, загружаемый операционной системой.

Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтверждение разрешает контроллеру издавать новые прерывания. Благодаря тому, что центральный процессор откладывает выдачу подтверждения до момента, когда он уже готов к обработке нового прерывания, удается избежать ситуации состязаний при появлении почти одновременных прерываний от нескольких устройств. Следует упомянуть, что на некоторых старых компьютерах нет микросхемы, централизованного контроллера прерываний, поэтому контроллер каждого устройства выставляет свое собственное прерывание.

Аппаратура всегда, прежде чем начать процедуру обработки прерывания, сохраняет определенную информацию. Сохраняемая информация и место ее хранения широко варьируются в зависимости от центрального процессора. Как минимум сохраняется счетчик команд, что позволяет продолжить выполнение прерванного процесса. Другая крайность представляет собой сохранение всех программно доступных регистров и большого количества внутренних регистров центрального процессора.

Место сохранения этой информации также оказывается проблемой. Один из вариантов состоит в том, чтобы сохранять эти данные в неких внутренних регистрах, доступных операционной системе. Недостаток такого подхода – до тех пор, пока вся сохраненная информация не будет считана обработчиком прерываний, новые прерывания будет нельзя разрешать. В противном случае любое новое прерывание просто стерло бы всю сохраненную таким образом информацию, записав поверх нее новые данные. В результате прерывания оказываются запрещенными в течение довольно длительных интервалов времени, что приводит к возможному игнорированию некоторых сигналов прерывания от устройств и, соответственно, к возможной потере данных.

Поэтому большинство центральных процессоров сохраняют информацию в стеке. Однако у этого подхода также имеются недостатки. Во-первых, в чьем стеке следует сохранять данные? Если использовать текущий стек, он может оказаться стеком процесса пользователя. При этом может даже выясниться, что пользователь использует указатель стека в своей программе весьма нестандартно, то есть стек может указывать на область памяти, в которой нельзя сохранять данные. Попытка записать несколько слов в стек в таком случае может привести к неисправимой ошибке. Также указатель стека может указывать на конец страницы памяти. После нескольких обращений к стеку указатель может достичь конца страницы памяти и вызвать соответствующую ошибку. Если во время обработки аппаратно-

го прерывания произойдет обращение к отсутствующей странице, это станет еще большей проблемой: где сохранить состояние процедуры обработки прерывания для обработки данной ошибки?

Использовать стек ядра гораздо проще, так как при этом больше шансов, что указатель стека указывает на область памяти, в которой можно сохранять данные. Однако переключение в режим ядра может потребовать изменения контекста MMU и, возможно, обесценит большую часть содержимого кэша и TLB (Translation Lookaside Buffer — буфер быстрого преобразования адреса). Перегрузка всех этих кэшей статически или динамически увеличит время обработки прерывания и вызовет растрату процессорного времени.

Другая проблема вызвана тем фактом, что большинство современных центральных процессоров широко используют конвейеры и часто являются суперскалярными (внутренне параллельными). В более старых системах после выполнения каждой команды процессора микропрограмма или аппаратура проверяли, нет ли прерывания, ждущего обработки. Если таковое было, счетчик команд и слово состояния процессора (PSW) сохранялись в стеке и начиналась обработка прерывания. По завершении работы обработчика прерывания происходила обратная процедура: старые значения PSW и счетчика команд извлекались из стека, и прерванный процесс возобновлялся.

Такая модель явно предполагает, что к приходу прерывания все команды процессора до этого момента выполнены полностью и ни одна команда процессора после последней выполненной команды не начала выполняться. На старых машинах такое предположение было оправдано. Однако на современных компьютерах это не всегда так.

Для начала рассмотрим модель конвейера на рис. 1.6, *а*. Что произойдет, если прерывание придет, когда конвейер полон (вполне обычный случай)? Различные команды окажутся на разных стадиях выполнения. К приходу прерывания значение счетчика команд может не отражать истинной границы между уже выполненными и еще не выполненными командами. Скорее всего, он будет указывать на адрес очередной команды, которую следует выбрать из памяти и поместить в конвейер, а не на адрес команды, только что обработанной исполнительным блоком.

В результате даже при наличии строго очерченной границы между уже выполненными и еще не выполненными командами может оказаться, что аппаратура просто не знает, где она проходит. Соответственно, при возврате из прерывания операционная система не может просто начать заполнять конвейер с адреса, содержащегося в счетчике команд. Она должна сначала узнать, какая команда была выполнена последней, что часто является довольно сложной задачей, требующей серьезного анализа состояния процессора.

Хотя эта ситуация и неприятна, однако прерывания на суперскалярной машине (рис. 1.6, *б*) значительно хуже. Поскольку команды процессора могут выполняться не в порядке их расположения в памяти, строго очерченной границы между уже выполненными и еще не выполненными командами может вообще не оказаться. Может, например, случиться, что команды 1, 2, 3, 5 и 8 уже выполнены, а команды 4, 6, 7, 9 и 10 — еще нет. Более того, счетчик команд теперь может указывать, например, на команды 9, 10 или 11.

Прерывание, оставляющее машину в строго определенном состоянии, называется **точным прерыванием** [353]. У такого прерывания четыре следующих свойства:

1. Счетчик команд (PC, Program Counter) сохраняется в известном месте.
2. Все команды до той, на которую указывает счетчик команд, выполнены полностью.
3. Ни одна команда после той, на которую указывает счетчик команд, не была выполнена.
4. Состояние команды, на которую указывает счетчик команд, известно.

Обратите внимание, что здесь не накладывается запрета на начало выполнения команд после той, на которую указывает счетчик команд. Утверждается лишь, что все изменения с памятью и регистрами процессора, произведенные благодаря началу выполнения этих инструкций, должны быть отменены прежде, чем начнется обработка сигнала прерывания процессором. Разрешается выполнение команды, на которую указывает счетчик команд. Также допускается ее невыполнение. Однако должно быть известно, выполнена она или нет. Часто случается, если прерывание пришло от устройства ввода-вывода, что выполнение команды, на которую указывает счетчик команд, еще и не начиналось. Однако если прерывание было эмулировано или вызвано обращением к отсутствующей странице, тогда счетчик команд обычно указывает на команду, вызвавшую прерывание.

Прерывание, не удовлетворяющее данным требованиям, называется **неточным прерыванием**. Такое прерывание крайне портит жизнь программистам, пишущим операционную систему, которым приходится в таком случае выяснять, что случилось и чему еще предстоит случиться. Машины с неточным прерыванием обычно в случае прерывания выгружают в стек огромное количество данных, чтобы дать операционной системе возможность определить, что происходило в этот момент. Сохранение больших объемов данных в памяти при каждом прерывании сильно замедляет вход в процедуру обработки прерывания, а восстановление после прерывания усложняется еще больше. Все это приводит к нелепой ситуации, в которой сверхбыстрый суперскалярный центральный процессор оказывается непригоден для задач реального времени из-за своих страшно медленных прерываний.

Некоторые компьютеры спроектированы таким образом, что одни типы прерываний (аппаратных и эмулированных) оказываются точными, тогда как другие — неточными. Например, совсем не плохо, если прерывания от устройств ввода-вывода будут точными, а эмулированные прерывания и прерывания, вызванные программными ошибками, будут неточными, так как последние не требуют возобновления прерванных процессов. В некоторых машинах имеется специальный бит, установив который можно все прерывания сделать точными. Недостатком установки такого бита является то, что он вынуждает процессор тщательно регистрировать свои действия и сохранять значения регистров в специальных теневых регистрах, обеспечивая, таким образом, возможность произвести точное прерывание в любой момент времени. Естественно, все эти накладные расходы заметно снижают производительность.

Некоторые суперскалярные процессоры, такие как Pentium Pro и все его преемники, поддерживают точные прерывания для корректной работы на них программ, написанных для старых процессоров 386, 486 и Pentium I. (Первым супер-

скалярным процессором серии Intel x86 был процессор Pentium Pro; у процессора Pentium I было просто два конвейера.) Ценой за точные прерывания оказывается крайне сложная внутрипроцессорная логика прерываний. Эта логика должна гарантировать, что в случае прихода сигнала прерывания от контроллера прерываний все команды вплоть до определенной точки могли быть завершены и ни одна команда после последней выполненной не должна была оказать заметного эффекта на состояние машины. В данном случае ценой, которую приходилось платить за возможность точных прерываний на суперскалярном процессоре, являлось не время обработки прерывания, а сложность самого процессора и его разработки. Если бы можно было отказаться от необходимости поддержки точных прерываний, что делалось для обратной совместимости, сэкономленное место на кристалле можно было бы отдать под кэш, а это ускорило бы процессор. С другой стороны, наличие неточных прерываний значительно усложняет и замедляет операционную систему, так что трудно сказать, какой подход на самом деле лучше.

Принципы программного обеспечения ввода-вывода

Перейдем теперь от рассмотрения аппаратуры ввода-вывода к знакомству с программным обеспечением ввода-вывода. Сначала мы познакомимся с целями программного обеспечения ввода-вывода, а затем с различными способами выполнения операций ввода-вывода с точки зрения операционной системы.

Задачи программного обеспечения ввода-вывода

Ключевая концепция разработки программного обеспечения ввода-вывода известна как **независимость от устройств**. Эта концепция означает возможность написания программ, способных получать доступ к любому устройству ввода-вывода, без предварительного указания конкретного устройства. Например, программа, читающая данные из входного файла, должна с одинаковым успехом работать с файлом на дискете, жестком диске или компакт-диске. При этом не должны требоваться какие-либо изменения в программе. Например, должна быть возможность дать команду вроде

```
sort <input >output
```

и эта команда должна работать, независимо от того, что указано в качестве входного устройства — гибкий диск, IDE-диск, SCSI-диск или клавиатура. В качестве выходного устройства также с равным успехом может быть указан экран, файл на любом диске или принтер. Все проблемы, связанные с отличиями этих устройств, должна решать операционная система.

Тесно связан с концепцией независимости от устройств принцип **единообразного именования**. Имя файла или устройства должно быть просто текстовой строкой или целым числом и никоим образом не зависеть от физического устройства. В системе UNIX все диски могут быть произвольным образом интегрированы в иерархию файловой системы, так что пользователю не обязательно знать, какое

имя какому устройству соответствует. Например, гибкий диск может быть **смонтирован** поверх каталога */usr/ast/backup*, так что копирование файла в каталог */usr/ast/backup/monday* автоматически приведет к копированию файлов на гибкий диск. Таким образом, все файлы и устройства адресуются одним и тем же способом: по имени пути.

Другим важным аспектом программного обеспечения ввода-вывода является **обработка ошибок**. Ошибки должны обрабатываться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку чтения, он должен попытаться по возможности исправить эту ошибку сам. Если он не может это сделать, тогда эту ошибку должен обработать драйвер устройства, возможно, попытавшись прочитать этот блок еще раз. Многие ошибки бывают временными, как, например, ошибки чтения, вызванные пылинками на читающих головках. Такие ошибки часто исчезают при повторной попытке чтения блока. Только если нижний уровень не может сам справиться с проблемой, о ней следует информировать верхний уровень. Во многих случаях восстановление после ошибок может осуществляться на нижнем уровне, прозрачно для верхних уровней, то есть так, что верхние уровни даже не будут знать о наличии ошибок.

Еще одним ключевым вопросом является способ переноса данных: **синхронный** (блокирующий) против **асинхронного** (управляемого прерываниями). Большинство операций ввода-вывода на физическом уровне являются асинхронными — центральный процессор запускает перенос данных и отправляется заниматься чем-либо другим, пока не придет прерывание. Программы пользователя значительно легче написать, используя блокирующие операции ввода-вывода — после обращения к системному вызову read программа автоматически приостанавливается до тех пор, пока данные не появятся в буфере. Тем, чтобы операции ввода-вывода, в действительности являющиеся асинхронными, выглядели как блокирующие в программах пользователя, занимается операционная система.

Еще одним аспектом программного обеспечения ввода-вывода является **буферизация**. Часто данные, поступающие с устройства, не могут быть сохранены сразу там, куда они в конечном итоге направляются. Например, когда пакет приходит по сети, операционная система не знает, куда его поместить, пока не будет изучено его содержимое, для чего этот пакет нужно где-то временно сохранить. Кроме того, для многих устройств реального времени крайне важными оказываются параметры сроков поступления данных (например для устройств воспроизведения цифрового звука), поэтому полученные данные должны быть помещены в выходной буфер заранее, чтобы скорость, с которой эти данные получаются из буфера воспроизводящей программой, не зависела от скорости заполнения буфера. Таким образом удается избежать неравномерности воспроизведения звука. Буферизация включает копирование данных в значительных количествах, что часто является основным фактором снижения производительности операций ввода-вывода.

И последним понятием, которое мы упомянем здесь, является понятие выделенных устройств и устройств коллективного использования. С некоторыми устройствами ввода-вывода, такими как диски, может одновременно работать большое количество пользователей. При этом не должно возникать проблем, если несколько пользователей на одном и том же диске одновременно откроют файлы. Другие устройства, такие как накопители на магнитной ленте, должны предоставляться

в монопольное владение одному пользователю, пока он не завершит свою работу с этим устройством. После этого накопитель может быть предоставлен другому пользователю. Если два или более пользователей одновременно станут писать вперемешку блоки на одну ленту, то ничего хорошего не получится. Введение понятия выделенных (монопольно используемых) устройств также привносит целый спектр проблем, например, как взаимоблокировки. Тем не менее операционная система должна уметь управлять как устройствами общего доступа, так и выделенными устройствами, позволяя избегать различных потенциальных проблем.

Программный ввод-вывод

Существует три фундаментально различных способа осуществления операций ввода-вывода. В этом разделе мы рассмотрим первый способ (программный ввод-вывод). В следующих двух разделах мы познакомимся с остальными разновидностями ввода-вывода (с управляемым прерываниями вводом-выводом и вводом-выводом с использованием DMA). Простейший вид ввода-вывода состоит в том, что всю работу выполняет центральный процессор. Этот метод называется **программным вводом-выводом**.

Проще всего проиллюстрировать программный ввод-вывод на примере. Рассмотрим процесс пользователя, которому нужно напечатать на принтере строку из восьми символов «ABCDEFGH». Сначала он собирает эту строку в буфере в пространстве пользователя (рис. 5.5, *а*).

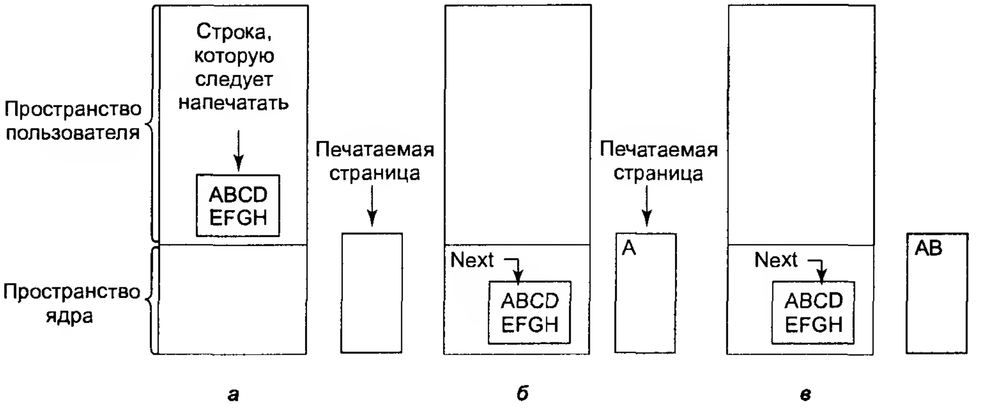


Рис. 5.5. Этапы печати строки

Затем, обращаясь к системному вызову, процесс пользователя получает принтер во временное пользование. Если принтер в данный момент оказывается занят другим процессом, обращение к системному вызову на открытие принтера завершится неудачей. Вызывающему процессу либо будет возвращен код ошибки, либо этот процесс будет блокирован до тех пор, пока принтер не освободится, в зависимости от операционной системы и параметров вызова. Получив принтер, процесс пользователя обращается к другому системному вызову, прося операционную систему распечатать строку на принтере.

Операционная система при этом обычно копирует содержимое буфера со строкой в некий массив, расположенный в пространстве ядра, где ей проще получить к этим данным доступ (поскольку ядру для получения доступа к пространству пользователя, возможно, придется изменять карту памяти). Затем она проверяет, доступен ли в данный момент принтер. Если нет, она ждет его освобождения. Как только принтер становится доступен, операционная система копирует первый символ в регистр данных принтера, используя в данном примере отображение регистров устройств ввода-вывода на память. Это действие активизирует принтер. На бумаге этот символ может сразу не появиться, так как большинство принтеров буферизируют целую строку или даже страницу данных прежде, чем начать собственно печать. Однако на рис. 5.5, *б* мы видим, что первый символ напечатан, а указатель операционной системы установлен на следующий символ (B).

Напечатав первый символ на принтере, операционная система проверяет, готов ли принтер к приему следующего символа. Обычно у принтера есть второй регистр, в котором можно прочитать его состояние. При записи символа в регистр данных принтер инвертирует бит готовности в статусном регистре. По окончании обработки полученного символа контроллером принтера бит готовности снова устанавливается, показывая, что принтер готов к приему следующего символа.

Итак, операционная система ждет, когда принтер снова перейдет в состояние готовности. Когда это происходит, она печатает следующий символ (рис. 5.5, *в*). Этот цикл продолжается до тех пор, пока не будет распечатана вся строка. После этого управление возвращается процессу пользователя.

Действия, выполняемые операционной системой, в виде программы на языке C продемонстрированы в листинге 5.1. Сначала данные копируются в ядро. Затем операционная система входит в цикл, в котором на каждой итерации цикла печатает на принтере один символ. Существенный аспект программного ввода-вывода, ясно проиллюстрированный данным примером, состоит в том, что после печати каждого символа процессор в цикле опрашивает готовность устройства. Такое поведение процессора называется **опросом** или **ожиданием готовности**, а также **активным ожиданием**.

Листинг 5.1. Печать строки при помощи программного ввода-вывода

```
copy_from_user(buffer, p, count);           /* p - буфер ядра */
for (i=0; i<count; i++){                    /* цикл символов */
      while (*printer_status_reg != READY); /* цикл ожидания готовности */
      * printer_data_reg = p[i];            /* печать символа */
}
return_to_user();
```

Программный ввод-вывод очень легко реализуется, но его существенный недостаток состоит в том, что центральный процессор занимается на все время операции ввода-вывода. Даже если один символ «печатается» очень быстро, поскольку все, что нужно сделать принтеру — это поместить этот символ в свой внутренний буфер, принтер обычно не рассчитан на прием символов с той скоростью, с которой их может выдать быстрый процессор. Поэтому большую часть времени центральный процессор проведет в ожидании готовности принтера, что является неэффективным использованием процессорного времени. Такой подход вполне допустим в примитивных встроенных системах, в которых у центрального процессора нет других задач; однако в более сложных, многозадачных системах такой подход неприемлем.

Управляемый прерываниями ввод-вывод

Рассмотрим теперь случай принтера, не буферизирующего символы, а печатающего их сразу по прибытии. Если принтер может печатать, скажем, 100 символов в секунду, то на печать каждого символа уходит 10 мс. Это значит, что после записи каждого символа в регистр данных принтера центральный процессор должен ждать в цикле целых 10 мс, пока ему не позволят записать в регистр следующий символ. Этого времени более чем достаточно для переключения контекста и запуска другого процесса на 10 мс, которые в противном случае просто будут потеряны.

Предоставить центральному процессору возможность делать что-нибудь в то время, когда принтер переходит в состояние готовности, можно при помощи прерываний. Когда выполняется системный вызов печати строки, как мы уже показывали, буфер копируется в пространство ядра и первый символ строки копируется на принтер, как только принтер выставит бит готовности. После этого центральный процессор вызывает планировщик, который запускает какой-либо другой процесс. Процесс, попросивший распечатать строку, оказывается заблокирован на весь период печати строки. Работа, выполняемая при системном вызове, показана на рис. 5.6, *а*.

```
copy_from_user(buffer,p,count);
enable_interrupts();
while(*printer_status_reg!=READY);
*printer_data_register=p[0];
scheduler();
```

а

```
if (count==0) {
    unblock_user();
} else {
        *printer_data_register=p[i];
        count=count-1;
        i=i+1;
}
acknowledge_interrupt();
return_from_interrupt();
```

б

Рис. 5.6. Печать строки при помощи ввода-вывода, управляемого прерываниями: программа, выполняемая при обращении к системному вызову (*а*); процедура обработки прерываний (*б*)

Когда принтер напечатал символ и готов принять следующий, он инициирует прерывание. Это прерывание вызывает остановку текущего процесса и сохранение его состояния. Затем запускается процедура обработки прерывания от принтера. Грубый вариант этой программы показан на рис. 5.6, *б*. Если напечатаны все символы, обработчик прерывания предпринимает необходимые меры для разблокировки процесса пользователя. В противном случае он печатает следующий символ, подтверждает прерывание и возвращается к процессу, выполнение которого было приостановлено прерыванием от принтера.

Ввод-вывод с использованием DMA

Очевидный недостаток управляемого прерываниями ввода-вывода состоит в том, что прерывания происходят при печати каждого символа. Обработка прерываний занимает определенное время, поэтому такая схема не является эффективной. Решение этой проблемы заключается в использовании DMA. Идея состоит в том, чтобы позволить контроллеру DMA поставлять принтеру символы по одному, не

беспокоя при этом центральный процессор. По существу, этот метод почти не отличается от программного ввода-вывода, с той лишь разницей, что всю работу вместо центрального процессора выполняет контроллер DMA. Набросок программы показан на рис. 5.7.

```
copy_from_user(buffer,p,count);
set_up_DMA_controller();
scheduler();
```

а

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

б

Рис. 5.7. Печать строки при помощи DMA: программа, выполняемая при обращении к системному вызову (*а*); процедура обработки прерываний (*б*)

Наибольший выигрыш от использования DMA состоит в уменьшении количества прерываний с одного на печатаемый символ до одного на печатаемый буфер. Если символов много, а прерывания обрабатываются медленно, то этот выигрыш весьма существенен. С другой стороны, контроллер DMA обычно значительно уступает центральному процессору в скорости. Если контроллер DMA не может поддерживать полную скорость ввода или вывода с внешнего устройства, либо у центрального процессора нет других задач во время ожидания прерывания от DMA, тогда оба предыдущих метода ввода-вывода (программный и управляемый прерываниями) будут предпочтительнее.

Программные уровни ввода-вывода

Программное обеспечение ввода-вывода обычно организуется в виде четырех уровней, показанных на рис. 5.8. У каждого уровня есть четко очерченная функция, которую он должен выполнять, и строго определенный интерфейс с соседними уровнями. Функции и интерфейсы уровней меняются от одной операционной системы к другой, поэтому последующее рассмотрение всех уровней, начиная с нижнего, не является специфичным для какой-либо конкретной машины.

Обработчики прерываний

Хотя программный ввод-вывод иногда бывает полезен, для большинства операций ввода-вывода прерывания являются неприятным, но необходимым фактом. Прерывания должны быть упрятаны как можно глубже во внутренностях операционной системы, чтобы о них знала как можно меньшая часть операционной системы. Лучший способ спрятать их заключается в блокировке драйвера, начавшего операцию ввода-вывода, вплоть до окончания этой операции и получения прерывания. Драйвер может заблокировать себя сам, выполнив на семафоре процедуру `down`, процедуру `wait` на переменной состояния, процедуру `receive` на сообщении, или что-либо подобное.

Когда происходит прерывание, начинает работу обработчик прерываний. По окончании необходимой работы он может разблокировать драйвер, запустивший его. В некоторых случаях используется выполнение процедуры `up` на семафоре.

В других случаях обработчик прерываний вызывает процедуру монитора `signal` с переменной состояния. В третьем случае он посылает заблокированному драйверу сообщение. В любом случае драйвер разблокируется обработчиком прерываний. Эта схема лучше всего работает в драйверах, являющихся процессами ядра со своим собственным состоянием, стеком и счетчиком команд.

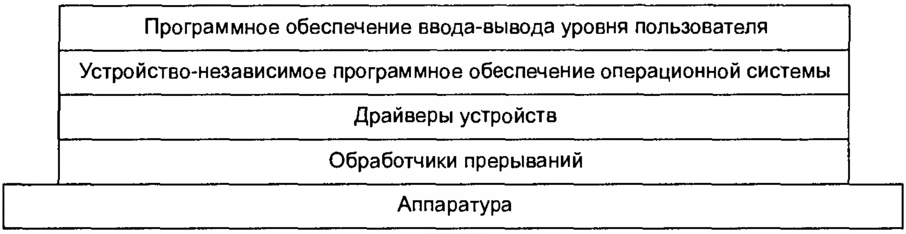


Рис. 5.8. Программные уровни ввода-вывода

Конечно, в действительности все обстоит совсем не так просто. Обработать прерывание значительно сложнее, чем просто принять его, выполнить `up` на семафоре, после чего вернуться из прерывания в предыдущий процесс с помощью команды процессора IRET. Операционной системе приходится выполнить значительно больше работы. Мы покажем схематичный набросок этой работы в виде набора шагов, которые следует выполнить программному обеспечению после того, как произошло аппаратное прерывание. Необходимо заметить, что детали во многом зависят от конкретной системы, поэтому на каких-то машинах некоторые перечисленные шаги могут оказаться лишними, зато может потребоваться выполнение других, не помещенных в список шагов. Кроме того, на разных машинах может потребоваться выполнение перечисленных действий в разном порядке.

1. Сохранить все регистры (включая PSW), не сохраненные аппаратурой.
2. Установить контекст для процедуры обработки прерываний. Выполнение этого действия может включать установку TLB, MMU и таблицы страниц.
3. Установить указатель стека для процедуры обработки прерываний.
4. Выдать подтверждение контроллеру прерываний. Если централизованного контроллера прерываний нет, разрешить прерывания.
5. Скопировать содержимое регистров с того места, где они были сохранены (возможно, в каком-либо стеке), в таблицу процессов.
6. Запустить процедуру обработки прерываний. Она извлечет информацию из регистров контроллера устройства, инициировавшего прерывание.
7. Выбрать процесс, которому передать управление. Если прерывание разблокировало какой-либо высокоприоритетный процесс, он может быть выбран в качестве следующего.
8. Установить контекст MMU для следующего работающего процесса. Также может понадобиться определенная установка TLB.
9. Загрузить регистры нового процесса, включая его PSW.
10. Начать выполнение нового процесса.

Как можно заметить, обработка прерываний является далеко не простым делом. Она состоит из значительного количества команд процессора, особенно на машинах с виртуальной памятью, на которых необходимо восстанавливать состояние таблиц памяти или сохраненное состояние MMU (например, биты R и M). На некоторых машинах буфер быстрого преобразования адреса TLB и кэш центрального процессора также требуют управления при переключении режимов пользователя и ядра, для чего необходимы дополнительные машинные циклы.

Драйверы устройств

Ранее в этой главе мы познакомились с функциями контроллеров устройств ввода-вывода. Как было сказано, у каждого контроллера есть набор регистров, используемых для того, чтобы давать управляемому им устройству команды и читать состояние устройства. Число таких регистров и команды, выдаваемые устройствам, зависят от конкретного устройства. Например, драйвер мыши должен принимать от мыши информацию о том, насколько далеко она продвинулась по горизонтали и вертикали, а также о нажатых кнопках мыши. Драйвер диска, в отличие от драйвера мыши, должен знать о секторах, дорожках, цилиндрах, головках, их перемещении и времени установки, двигателях и тому подобных вещах, необходимых для правильной работы диска. Очевидно, что эти драйверы будут сильно различаться.

Поэтому для управления каждым устройством ввода-вывода, подключенным к компьютеру, требуется специальная программа. Эта программа, называемая **драйвером устройства**, обычно пишется производителем устройства и распространяется вместе с устройством. Поскольку для каждой операционной системы требуются специальные драйверы, производители устройств обычно поставляют драйверы для нескольких наиболее популярных операционных систем.

Каждый драйвер устройства обычно поддерживает один тип устройств или, максимум, класс близких устройств. Например, драйвер SCSI-дисков обычно может поддерживать различные SCSI-диски, отличающиеся размерами и скоростями, и возможно даже будет поддерживать SCSI CD-ROM. С другой стороны, мышь и джойстик отличаются настолько сильно, что обычно требуют использования различных драйверов. Однако нет никакого технического ограничения на управление одним драйвером нескольких непохожих устройств. Это просто не слишком удачная идея.

Чтобы получить доступ к аппаратной части устройства, то есть к регистрам контроллера, драйвер устройства должен быть частью ядра операционной системы, по крайней мере, в существующих на сегодняшний день архитектурах. В действительности возможно создать и драйвер, работающий в пространстве пользователя, с системными вызовами для чтения и записи регистров устройств. В самом деле, это было бы даже неплохой идеей, так как позволило бы изолировать ядро от драйверов, а драйверы друг от друга. При этом была бы устранена основная причина крушения операционной системы — драйверы, содержащие ошибки, сталкивающиеся с ядром тем или иным образом. Тем не менее, поскольку современные операционные системы предполагают работу драйверов в ядре, мы рассмотрим здесь именно такую модель.

Так как в операционную систему будут устанавливаться куски программ (драйверы), написанные другими программистами, необходима определенная архитектура, позволяющая подобную установку. Это означает, что должна быть выработана строго определенная модель функций драйвера и его взаимодействия с остальной операционной системой. Драйверы устройств обычно располагаются под остальной операционной системой, как показано на рис. 5.9.

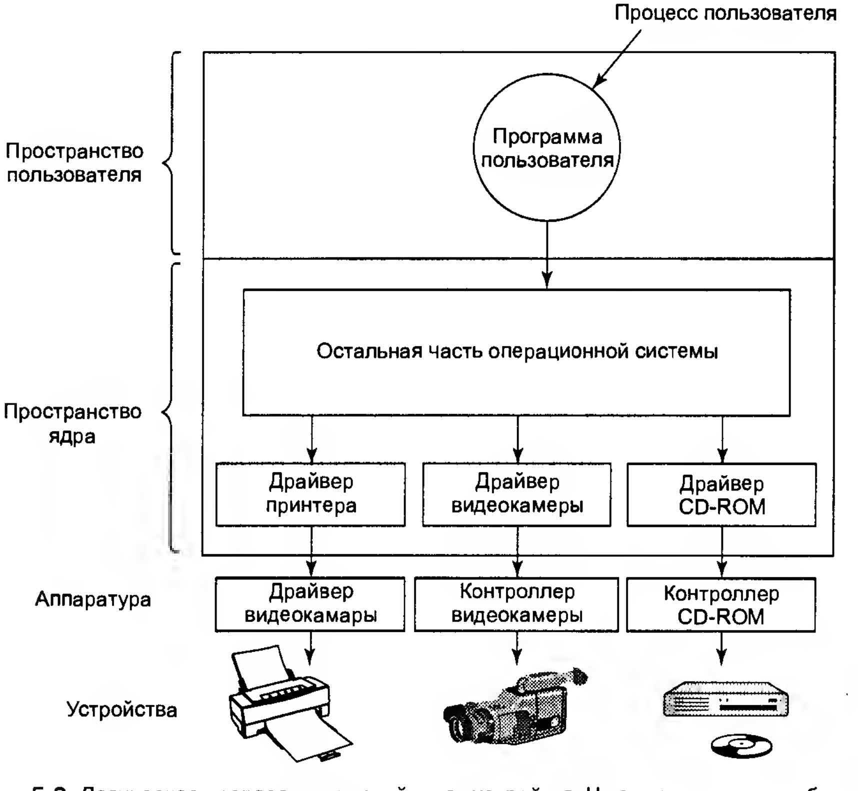


Рис. 5.9. Логическое расположение драйверов устройств. На самом деле весь обмен информацией между драйверами и контроллерами устройств идет по шине

Операционная система обычно классифицирует драйверы по нескольким категориям в соответствии с типами обслуживаемых ими устройств. К наиболее общим категориям относятся **блочные устройства**, например, диски, содержащие блоки данных, к которым возможна независимая адресация, и **символьные устройства**, такие как клавиатуры и принтеры, формирующие или принимающие поток символов.

В большинстве операционных систем определен стандартный интерфейс, который должны поддерживать все блочные драйверы, и второй стандартный интерфейс, поддерживаемый всеми символьными драйверами. Эти интерфейсы включают наборы процедур, которые могут вызываться остальной операционной системой для обращения к драйверу. К этим процедурам относятся, например, процедуры чтения блока (блочного устройства) или записи символьной строки (для символьного устройства).

В некоторых системах операционная система представляет собой единую двоичную программу, содержащую в себе, в откомпилированном вместе с ней виде, все необходимые ей драйверы. Такая схема в течение многих лет была нормой для систем UNIX, так как они предназначались для работы в компьютерных центрах, а устройства ввода-вывода менялись нечасто. При добавлении нового устройства системный администратор просто перекомпилировал ядро с новым драйвером, получая новый двоичный модуль.

С появлением персональных компьютеров с их огромным разнообразием устройств ввода-вывода такая модель перестала работать. Далеко не все пользователи могли самостоятельно перекомпилировать и собрать ядро даже при наличии исходных текстов или объектных модулей, что, кстати, также не всегда имеет место. Вместо этого операционные системы, начиная с MS-DOS, перешли к модели динамической подгрузки драйверов во время выполнения системы. Различные системы загружают драйверы по-разному.

У драйвера устройства есть несколько функций. Наиболее очевидная функция драйвера состоит в обработке абстрактных запросов чтения и записи независимого от устройств программного обеспечения, расположенного над ними. Но кроме этого они должны также выполнять еще несколько функций. Например, драйвер должен при необходимости инициализировать устройство. Ему также может понадобиться управлять энергопотреблением устройства и регистрацией событий.

Многие драйверы устройств обладают сходной общей структурой. Типичный драйвер начинает с проверки входных параметров. Если они не удовлетворяют определенным критериям, драйвер возвращает ошибку. В противном случае драйвер преобразует абстрактные термины в конкретные. Например, дисковый драйвер может преобразовывать линейный номер блока в номера головки, дорожки и секторы.

Затем драйвер может проверить, не используется ли это устройство в данный момент. Если устройство занято, запрос может быть поставлен в очередь. Если устройство свободно, проверяется аппаратный статус устройства, чтобы понять, может ли запрос быть обслужен прямо сейчас. Может оказаться необходимым включить устройство или запустить двигатель, прежде чем начнется перенос данных. Как только устройство включено и готово, может начинаться собственно управление устройством.

Управление устройством подразумевает выдачу ему серии команд. Именно в драйвере определяется последовательность команд в зависимости от того, что должно быть сделано. Определившись с командами, драйвер начинает записывать их в регистры контроллера устройства. После записи каждой команды в контроллер может быть нужно проверить, принял ли контроллер эту команду и готов ли принять следующую. Такая последовательность действий продолжается до тех пор, пока контроллеру не будут даны все команды. Некоторые контроллеры способны принимать связные списки команд, находящихся в памяти. Они сами считывают и выполняют их без дальнейшей помощи операционной системы.

После того как драйвер передал все команды контроллеру, ситуация может развиваться по двум сценариям. Во многих случаях драйвер устройства должен ждать, пока контроллер не выполнит для него определенную работу, поэтому он блокируется до тех пор, пока прерывание от устройства его не разблокирует. В других случаях операция завершается без задержек и драйверу не нужно блокироваться. Например, для скроллинга экрана в символьном режиме нужно записать

всего лишь несколько байтов в регистры контроллера. Вся операция занимает несколько наносекунд.

В первом случае заблокированный драйвер будет активизирован прерыванием. Во втором случае драйвер не блокируется. В любом случае по завершении выполнения операции драйвер должен проверить, завершилась ли операция без ошибок. Если все в порядке, драйверу, возможно, придется предать данные (например, только что прочитанный блок) независимому от устройств программному обеспечению. Наконец, драйвер возвращает некоторую информацию о состоянии для информирования вызывающей программы о статусе завершения операции. Если в очереди находились другие запросы, один из них теперь может быть выбран и запущен. В противном случае драйвер блокируется в ожидании следующего запроса.

Эта упрощенная модель является лишь грубым приближением к реальности. На самом деле программа значительно сложнее, причиной чему служит большое количество разнообразных факторов. Во-первых, устройство ввода-вывода может завершить выполнение операции во время работы драйвера, таким образом прерывая его работу. Во-вторых, во время обработки сетевым драйвером пришедшего пакета может прийти еще один пакет. Соответственно, драйвер должен быть **реентерабельным**, то есть должен быть готов к тому, что во время обработки первого вызова может последовать другой вызов.

В системе с возможностью горячей установки устройства могут добавляться или удаляться во время работы системы. В результате в то время, когда драйвер занят чтением с какого-либо устройства, система может проинформировать его, что пользователь внезапно удалил это устройство из системы. При этом не только текущая операция переноса данных должна быть прервана без повреждения структур данных ядра, но также и все ожидающие обработки запросы к теперь исчезнувшему устройству должны быть корректно удалены из системы, а обратившимся к ним программам должна быть сообщена неприятная новость. Более того, неожиданное добавление нового устройства может заставить ядро жонглировать ресурсами (например, линиями запросов прерывания), отнимая их у одного драйвера и предоставляя другому драйверу.

Драйверам не разрешается обращаться к системным вызовам, но им часто бывает необходимо взаимодействовать с остальным ядром. Обычно разрешаются обращения к некоторым системным процедурам. Например, драйверы обращаются к системным процедурам для выделения им аппаратно фиксированных страниц памяти в качестве буферов, а также затем, чтобы вернуть эти страницы обратно ядру. Кроме того, драйверы пользуются вызовами, управляющими MMU, таймерами, контроллером DMA, контроллером прерываний и т. п.

Независимое от устройств программное обеспечение ввода-вывода

Хотя некоторая часть программного обеспечения ввода-вывода предназначена для работы с конкретными устройствами, другая часть является независимой от устройств. Расположение точной границы между драйверами и независимым от устройств программным обеспечением зависит от системы (и устройств), так как некоторые функции, которые могут быть выполнены независимым от устройств

образом, часто выполняются прямо в драйверах из различных соображений, в том числе соображений эффективности. Функции, показанные в табл. 5.2, обычно реализуются в независимом от устройств программном обеспечении.

Таблица 5.2. Функции независимого от устройств программного обеспечения

Единообразный интерфейс для драйверов устройств
Буферизация
Сообщение об ошибках
Захват и освобождение выделенных устройств
Размер блока, не зависящий от устройства

Основная задача независимого от устройств программного обеспечения состоит в выполнении функций ввода-вывода, общих для всех устройств, и предоставлении единообразного интерфейса для программ уровня пользователя. Ниже мы рассмотрим некоторые из этих вопросов более подробно.

Единообразный интерфейс для драйверов устройств

Главный вопрос операционной системы — как сделать так, чтобы все устройства ввода-вывода и драйверы выглядели более-менее одинаково. Если диски, принтеры, клавиатуры и т. д. требуют различных интерфейсов, при появлении каждого нового устройства будет требоваться переделка операционной системы, что определенно не является удачной мыслью.

Этот вопрос связан с интерфейсом между драйверами устройств и остальной операционной системой. На рис. 5.10, *а* показана ситуация, в которой у всех драйверов различный интерфейс с операционной системой. Это означает, что функции драйвера, доступные системе, отличаются от драйвера к драйверу. Это может также означать, что функции ядра, необходимые для драйвера, тоже различаются. Все вместе это означает, что взаимодействие с каждым новым драйвером требует больших усилий программистов.

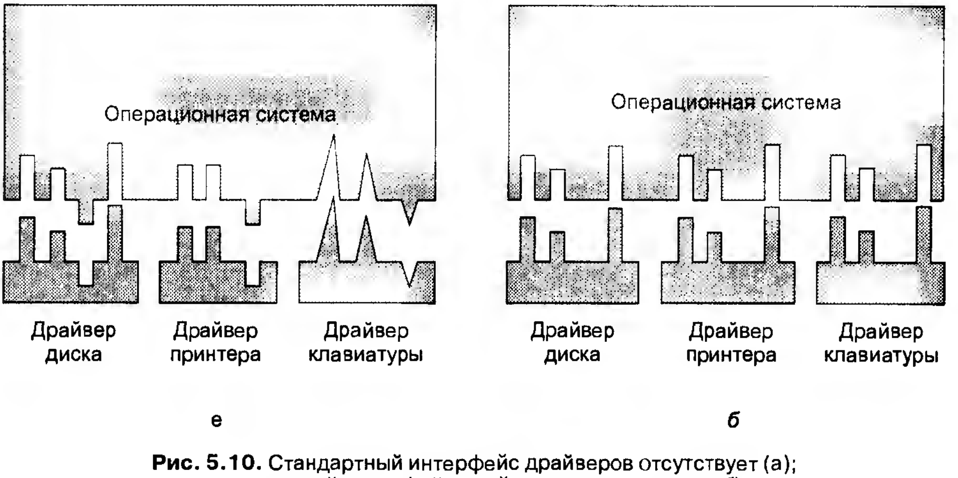


Рис. 5.10. Стандартный интерфейс драйверов отсутствует (а); стандартный интерфейс драйверов присутствует (*б*)

Напротив, на рис. 5.10, *б* изображен принципиально другой подход, при котором у всех драйверов один и тот же интерфейс. При этом значительно легче установить новый драйвер, при условии, что он соответствует стандартному интерфейсу. Это также означает, что программисты, пишущие драйверы, знают, чего от них ожидают (то есть какие функции они должны реализовать и к каким функциям ядра они могут обращаться). На практике не все устройства являются абсолютно идентичными, но обычно имеется небольшое число типов устройств, достаточно сходных друг с другом. Например, даже у блочных и символьных устройств есть много общих функций.

Другой аспект единообразного интерфейса состоит в именовании устройств ввода-вывода. Независимое от устройств программное обеспечение занимается отображением символьных имен устройств на соответствующие драйверы. Например, в системе UNIX имя устройства */dev/disk0* однозначно указывает i-узел специального файла, содержащий **номер главного устройства**, использующийся для определения расположения подходящего драйвера. Этот i-узел также содержит **номер второстепенного устройства**, передаваемый в виде параметра драйверу для указания конкретного диска или раздела диска, к которому относится операция чтения или записи. Все устройства в системе UNIX имеют главный и второстепенный номера, по которым они однозначно идентифицируются. Выбор всех драйверов осуществляется по главному номеру устройства.

С именованием устройств тесно связан вопрос защиты. Как операционная система предотвращает доступ пользователей к устройствам, на который у них нет прав? В UNIX и в Windows 2000 устройства представляются в файловой системе в виде именованных объектов, что дает возможность применять обычные правила защиты файлов к устройствам ввода-вывода. Таким образом, системный администратор может установить нужные разрешения для каждого устройства.

Буферизация

Буферизация также является важным вопросом как для блочных, так и для символьных устройств по самым разным причинам. Чтобы проиллюстрировать одну из причин, рассмотрим процесс, который хочет прочитать данные с модема. Одна из возможных стратегий обработки поступающих символов состоит в обращении процесса пользователя к системному вызову read и блокировке в ожидании отдельного символа. Каждый прибывающий символ вызывает прерывание. Процедура обработки прерываний передает символ пользовательскому процессу и разблокирует его. Поместив куда-либо полученный символ, процесс читает следующий символ, опять блокируясь. Эта схема проиллюстрирована на рис. 5.11, *а*.

Недостаток такого подхода состоит в том, что процесс пользователя должен быть активирован при прибытии каждого символа, что крайне неэффективно.

Улучшенный вариант показан на рис. 5.11, *б*. Здесь пользовательский процесс предоставляет буфер размером в *n* символов в пространстве пользователя, после чего выполняет чтение *n* символов. Процедура обработки прерываний помещает приходящие символы в буфер до тех пор, пока он не заполнится. Затем она активизирует процесс пользователя. Такая схема гораздо эффективнее предыдущей, однако у нее также есть недостатки: что случится, если в момент прибытия символа страница памяти, в которой расположен буфер, окажется выгруженной из физичес-

кой памяти? Конечно, буфер можно зафиксировать в памяти; если слишком много процессов начнут фиксировать свои страницы в памяти, пул доступных страниц памяти уменьшится, в результате чего снизится производительность.

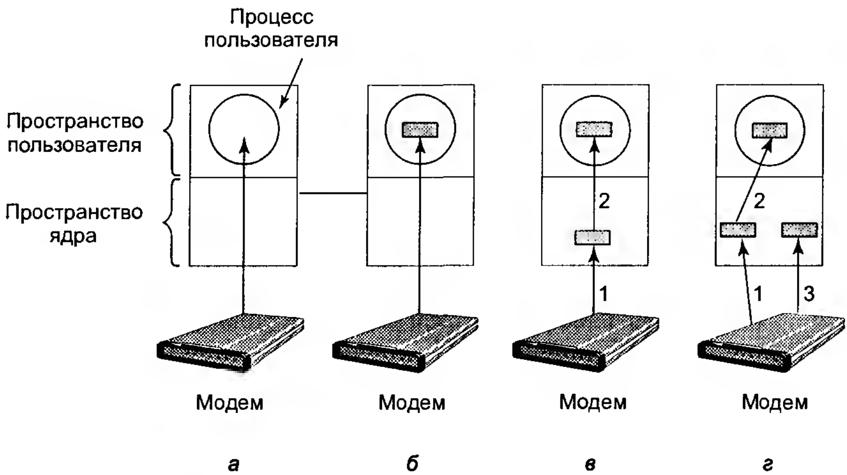


Рис. 5.11. Небуферизированный ввод (*а*); буферизация в пространстве пользователя (*б*); буферизация в ядре с копированием в пространство пользователя (*в*); двойная буферизация в ядре (*г*)

Следующий подход состоит в создании буфера, в который обработчик прерываний будет помещать поступающие символы, в ядре, как показано на рис. 5.11, *в*. Когда этот буфер наполняется, достается страница с буфером пользователя, и содержимое буфера копируется туда за одну операцию. Такая схема намного эффективнее.

Однако даже при использовании этой схемы имеются определенные проблемы. Что случится с символами, прибывающими в тот момент, когда страница пользователя загружается с диска? Поскольку буфер полон, их некуда поместить. Решение проблемы состоит в использовании второго буфера в ядре, в который помещаются символы после заполнения первого буфера до его освобождения (рис. 5.11, *г*). При этом буферы как бы меняются местами, то есть первый буфер начинает играть роль запасного. Такая схема часто называется **двойной буферизацией**.

Буферизация также играет важную роль при выводе данных. Рассмотрим, например, как происходит вывод на модем при помощи модели, показанной на рис. 5.11, *б*. Пользовательский процесс выполняет системный вызов `write` для вывода *n* символов. У системы в этот момент есть выбор. Она может заблокировать пользователя, пока все символы не будут записаны, но по медленной телефонной линии это может занять довольно много времени. Она может разрешить пользователю продолжать выполнение немедленно и выполнять операцию ввода-вывода, в то время как пользователь занимается другими вычислениями. Однако при этом возникает непростая проблема: как пользовательский процесс узнает, что операция вывода завершена и он может опять пользоваться этим буфером? Система может подать сигнал или программное прерывание, но такой стиль программирования сложен и чреват возникновениями ситуаций состязания. Значительно лучшее решение состоит в копировании данных в буфер ядра, по аналогии

с рис. 5.11, *в* (но в обратном направлении), и немедленном разблокировании вызывающего процесса. Теперь уже не важно, когда будет выполнена физическая операция ввода-вывода. Пользователь может использовать буфер сразу после возврата ему управления.

Буферизация является широко применяемой технологией, однако у нее имеются свои недостатки. Если при буферизации данные копируются слишком много раз, страдает производительность. Рассмотрим, например, сеть (рис. 5.12). Пользователь обращается к системному вызову для записи данных по сети. Ядро копирует пакет в буфер ядра, чтобы пользователь мог немедленно продолжить работу (шаг *1*).

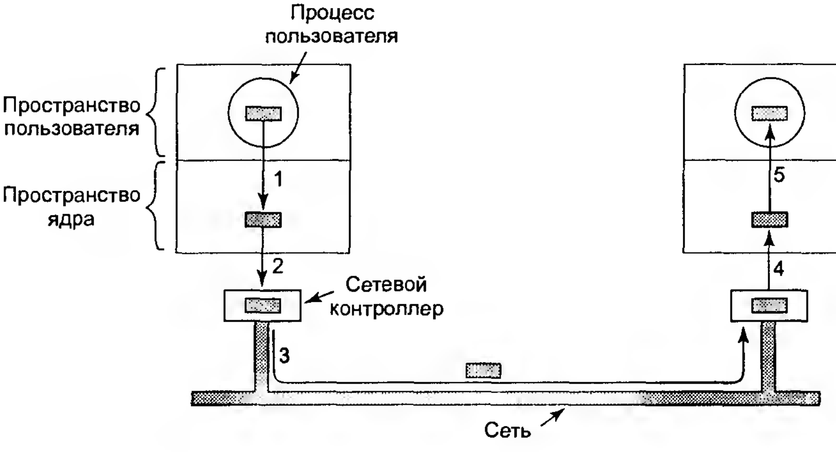


Рис. 5.12. Копии пакета при передаче его по сети

При вызове драйвера пакет копируется в контроллер для вывода (шаг *2*). Память ядра не используется напрямую, так как передача по линии должна производиться с постоянной скоростью. Драйвер не может гарантировать, что он сможет получать доступ к памяти с постоянной скоростью, так как каналы DMA и другие устройства ввода-вывода могут внезапно захватить много циклов шины. Если драйвер не сможет предоставить контроллеру вовремя хотя бы одно слово данных, весь пакет будет разрушен. Этой проблемы удается избежать при помощи буферизации пакета целиком внутри контроллера.

Затем пакет копируется в сеть (шаг *3*). Получающая сторона собирает пакет из отдельных битов также в буфере сетевого контроллера. После этого пакет копируется в буфер ядра получателя (шаг *4*). Наконец, он копируется в буфер получающего процесса (шаг *5*). Обычно в ответ получатель отправляет подтверждение. Получив подтверждение, отправитель может посылать следующий пакет. Однако должно быть очевидно, что все эти операции копирования существенно замедляют передачу данных, поскольку все эти шаги должны выполняться последовательно.

Сообщения об ошибках

Ошибки значительно чаще случаются в контексте ввода-вывода, нежели в других контекстах. При возникновении ошибок операционная система должна обработать их как можно быстрее. Многие ошибки являются специфичными для конкретно-

го устройства и должны обрабатываться соответствующим драйвером, но структура обработки ошибок является независимой от устройств.

Один из классов ошибок ввода-вывода составляют ошибки программирования. Они происходят, когда процесс просит чего-либо невозможного — например, записать данные на устройство ввода (клавиатуру, мышь, сканер и т. д.) или, наоборот, прочитать данные с устройства вывода (принтера, плоттера и т. п.). Другие ошибки включают предоставление неверного адреса буфера или иных параметров, а также обращение к несуществующему устройству (например, к диску 3, когда у системы всего два диска). Обработка таких ошибок довольно проста: код ошибки возвращается вызывающему процессу.

К другому классу ошибок относятся собственно ошибки ввода-вывода, такие как попытка записать в поврежденный блок диска или попытка прочитать данные с выключенной видеокамеры. В таких случаях драйвер сам решает, что ему делать. Если драйвер не может сам принять решение, он передает сведения о возникшей проблеме в вышестоящие инстанции (независимому от устройств программному обеспечению).

Действия этого программного обеспечения зависят от окружения и природы ошибки. Если это простая ошибка чтения, а программа интерактивная, можно отобразить диалоговое окно с вопросом к пользователю о дальнейших действиях. В качестве выбора может быть предложено повторение попытки определенное число раз, игнорирование ошибки или уничтожение процесса. Если программа не интерактивная, единственная возможность состоит в аварийном завершении системного вызова с соответствующим кодом ошибки.

Однако не все ошибки могут быть обработаны подобным образом. Например, может оказаться поврежденной критическая структура данных, такая как корневой каталог или список свободных блоков. В этом случае, возможно, системе придется вывести сообщение об ошибке и завершить свою работу.

Захват и освобождение выделенных устройств

Некоторые устройства, например устройство записи компакт-дисков, могут использоваться в каждый момент времени только одним пользователем. Операционная система должна рассмотреть запросы на использование этого устройства и либо принять их, либо отказать в выполнении запроса, в зависимости от доступности запрашиваемого устройства. Простой способ обработки этих запросов состоит в требовании к процессам обращаться напрямую к системному вызову open по отношению к специальным файлам для данных устройств. Если устройство недоступно, системный вызов open завершится неуспешно. Обращение к системному вызову close освобождает устройство.

Альтернативный подход состоит в предоставлении специального механизма для запроса и освобождения выделенных устройств. Попытка захватить недоступное устройство вызовет блокировку вызывающего процесса вместо возврата с ошибкой. Блокированные процессы устанавливаются в очередь. Раньше или позже запрашиваемое устройство освобождается и первому процессу в очереди разрешается захватить его и продолжить выполнение.

Независимый от устройств размер блока

У различных дисков могут быть разные размеры сектора. Независимое от устройств программное обеспечение должно скрывать этот факт от верхних уровней и предоставлять им единообразный размер блока, например, объединяя несколько физических сегментов в один логический блок. При этом более высокие уровни имеют дело только с абстрактными устройствами, с одним и тем же размером логического блока, не зависящим от размера физического сектора. Некоторые символьные устройства предоставляют свои данные побайтно (например, модемы), тогда как другие выдают их большими порциями (например, сетевые интерфейсы). Эти различия также могут быть скрыты.

Программное обеспечение ввода-вывода пространства пользователя

Хотя большая часть программного обеспечения ввода-вывода находится в операционной системе, небольшие его порции состоят из библиотек, присоединенных к программам пользователя, или даже целых программ, работающих вне ядра. Системные вызовы, включая системные вызовы ввода-вывода, обычно состоят из библиотечных процедур. Если программа на С содержит вызов

```
count = write(fd, buffer, nbytes);
```

библиотечная процедура *write* будет скомпонована с программой и, таким образом, будет содержаться в двоичной программе, загружаемой в память во время выполнения программы. Набор всех этих библиотечных процедур, несомненно, является частью системы ввода-вывода.

Хотя многие такие процедуры мало что делают, кроме обращения к системному вызову с соответствующими параметрами, есть ряд процедур ввода-вывода, выполняющих определенную работу. В частности, библиотечными процедурами выполняются операции форматного ввода и вывода. Например, процедура *printf* языка С, принимающая на входе текстовую строку и, возможно, несколько переменных, создает из нее ASCII-строку, после чего обращается к системному вызову `write` для вывода строки. Для примера рассмотрим следующий оператор

```
printf("Квадрат числа %3d равен %6d\n", i, i*i);
```

Он форматирует строку, состоящую из 14-символьной строки "Квадрат числа", за которой следует значение переменной i в виде 3-символьной строки, затем 7-символьной строки "равен", потом i^2 в виде 6 символов и, наконец, символа перевода строки.

Примером сходной процедуры ввода может служить *scanf*, читающая текстовую строку и преобразующая ее в значения переменных в соответствии с форматом, сходным с используемым процедурой *printf*. Стандартная библиотека ввода-вывода содержит большое количество процедур, включающих операции ввода-вывода и работающих как часть программы пользователя.

Не все программное обеспечение ввода-вывода пространства пользователя состоит из библиотечных процедур. Другую важную категорию составляет система спулинга. **Спулинг** (spooling — подкачка, предварительное накопление данных)

представляет собой способ работы с выделенными устройствами в многозадачной системе. Рассмотрим типичное устройство, на котором используется спулинг: принтер. В принципе можно разрешить каждому пользователю открывать специальный символьный файл принтера, однако представьте себе, что процесс открыл его, а затем не обращался к принтеру в течение нескольких часов. Ни один другой процесс в это время не сможет ничего напечатать.

Вместо этого создается специальный процесс, называемый **демоном**, и специальный каталог, называемый **каталогом спулинга** или **каталогом спулера**. Чтобы распечатать файл, процесс сначала создает специальный файл, предназначенный для печати, который помещает в каталог спулинга. Этот файл печатает демон, единственный процесс, которому разрешается пользоваться специальным файлом принтера. Таким образом, потенциальная проблема, связанная с тем, что какой-либо процесс на слишком долгий срок захватит принтер, решается при помощи защиты специального файла принтера от прямого доступа пользователей.

Спулинг используется не только для принтеров. Например, передачу файлов по сети также часто осуществляет специальный сетевой демон. Чтобы послать куда-либо файл, пользователь помещает его в каталог сетевого демона. Затем сетевой демон извлекает оттуда файл и посылает по сети. Подобный способ передачи файлов по сети используется системой сетевых новостей USENET News. Эта сеть состоит из миллионов машин по всему миру, общающихся друг с другом по Интернету. Существуют тысячи конференций по самым разным темам. Чтобы послать новое сообщение, пользователь вызывает программу новостей, которая принимает сообщение, а затем помещает его в каталог спулинга для последующей передачи на другие машины. Вся система новостей работает вне операционной системы.

На рис. 5.13 показана структура системы ввода-вывода, со всеми уровнями и основными функциями каждого уровня. Снизу вверх эти уровни представляют собой аппаратуру, обработчики прерываний, независимое от устройств программное обеспечение и, наконец, процессы пользователя.

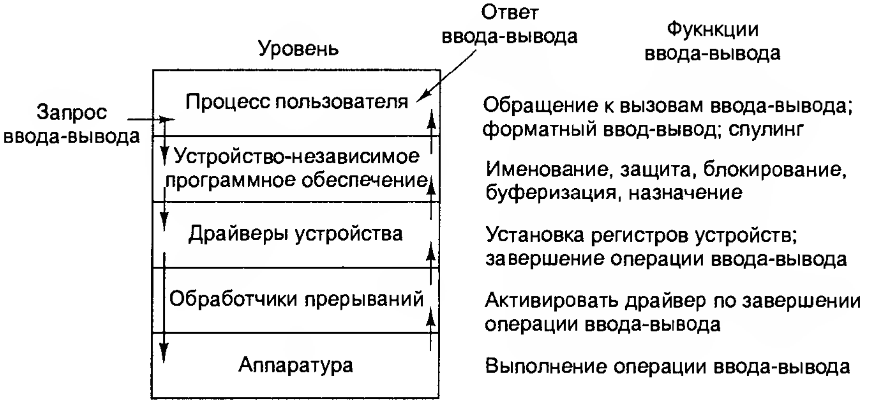


Рис. 5.13. Уровни и основные функции системы ввода-вывода

Стрелки на рис. 5.13 изображают поток управления. Например, когда программа пользователя пытается прочитать блок из файла, для обработки вызова запус-

кается операционная система. Независимое от устройств программное обеспечение ищет этот блок в кэше. Если требуемого блока там нет, оно вызывает драйвер устройства, чтобы обратиться к аппаратуре и получить этот блок с диска. При этом процесс блокируется до тех пор, пока не завершится дисковая операция.

Когда диск завершает операцию, аппаратура инициирует прерывание. Обработчик прерываний запускается, чтобы определить, что случилось, то есть какое устройство требует внимания. Затем он извлекает статус устройства и активизирует «спящий» процесс, чтобы завершить запрос ввода-вывода и предоставить пользовательскому процессу возможность продолжать.

Диски

Рассмотрим теперь некоторые реальные устройства ввода-вывода. Мы начнем с дисков. Затем мы также познакомимся с часами, клавиатурами и дисплеями.

Аппаратная часть дисков

Существует множество типов дисков. К наиболее часто встречающимся относятся магнитные диски (жесткие и гибкие). Их особенностью является одинаковая скорость чтения и записи, что делает их идеальными в качестве дополнительной памяти (страничная подкачка файлов, файловые системы и т. д.). Иногда, с целью создания высоконадежного устройства хранения, используются наборы жестких магнитных дисков. Для распространения программ, данных и фильмов используются различные виды оптических дисков (CD-ROM, CD-R, CD-RW и DVD). В следующих разделах мы сначала познакомимся с аппаратной частью, а затем с программным обеспечением для этих устройств.

Магнитные диски

Магнитные диски организованы в цилиндры, каждый из которых содержит столько дорожек, сколько есть у устройства головок, установленных вертикально. Дорожки делятся на секторы, их количество обычно варьируется от 8 до 32 у гибких дисков и до нескольких сот у жестких дисков. Число головок варьируется от 1 до 16.

У некоторых магнитных дисков мало электроники, они предоставляют на выходе простой поток битов. На этих дисках контроллер выполняет совсем немного работы. На других дисках, в частности на **IDE-дисках** (IDE, Integrated Drive Electronics — встроенный интерфейс накопителей), само устройство содержит микроконтроллер, выполняющий значительный объем работ, и позволяющий собственно контроллеру обращаться к нему с набором команд высокого уровня.

IDE-диски обладают одним свойством, обладающим важными последствиями для драйверов: контроллер способен выполнять одновременно поиск дорожки на двух и более дисках. Это свойство известно под названием **перекрывающегося поиска**. В то время как контроллер и программное обеспечение ожидают окончания операции поиска на одном устройстве, контроллер может инициировать поиск на другом устройстве. Многие контроллеры жестких дисков также могут совмещать

операцию чтения или записи на одном диске с поиском на другом или даже нескольких дисках. Однако контроллеры гибких дисков не могут одновременно читать и писать на двух дисководах. (Чтение или запись требует от контроллера перемещения битов с максимальной скоростью, на которую он рассчитан, поэтому операция чтения или записи использует большую часть его вычислительных мощностей.) В случае жестких дисков с их встроенными микроконтроллерами ситуация радикально меняется, поэтому несколько жестких дисков могут одновременно работать в одной системе, по крайней мере переносить данные между диском и буфером контроллера. Однако между контроллером и оперативной памятью в каждый момент времени может происходить только одна операция по переносу данных. Способность одновременного выполнения двух или более дисковых операций может существенно снизить среднее время доступа.

В табл. 5.3 сравниваются параметры стандартного средства оригинального компьютера IBM PC с параметрами современного жесткого диска для демонстрации того, насколько сильно изменились магнитные диски за последние два десятилетия. Интересно отметить, что не все параметры улучшились в равной мере. Среднее время поиска дорожки улучшилось в семь раз, скорость передачи данных увеличилась в 1300 раз, тогда как емкость диска увеличилась в 50 000 раз. Это различие вызвано относительно незначительными усовершенствованиями механической части по сравнению со значительно большим прогрессом в области увеличения плотности записи.

Таблица 5.3. Параметры 360-Кбайтного НГМД в сравнении с жестким диском Western Digital WD 18300

Параметр	НГМД IBM 360 Кбайт	Жесткий диск WD 18300
Количество цилиндров	40	10601
Дорожек в цилиндре	2	12
Секторов на дорожке	9	281 (среднее)
Секторов на диске	720	35 742 000
Байтов в секторе	512	512
Емкость диска	360 Кбайт	18,3 Гбайт
Время поиска (соседние цилиндры)	6 мс	0,8 мс
Время поиска (среднее)	77 мс	6,9 мс
Период обращения	200 мс	8,33 мс
Время запуска/остановки двигателя	250 мс	20 с
Время передачи 1 сектора	22 мс	17 мкс

Глядя на спецификации современных жестких дисков, следует помнить, что указанная геометрия, используемая программным обеспечением драйвера, может отличаться от физического формата. На старых дисках число секторов на дорожке было одинаково на всех цилиндрах. Современные диски разделены на зоны с большим числом секторов на внешних дорожках и меньшим на внутренних. На рис. 5.14, *а* изображен крошечный диск с двумя зонами. На внешней зоне каждая дорожка состоит из 32 секторов, тогда как на внутренней зоне по 16 секторов на дорожке. Реальные диски, как, например, WD 18300, часто имеют по 16 зон

с числом секторов, увеличивающимся примерно на 4 % в каждой следующей зоне при движении от центра диска к краю.

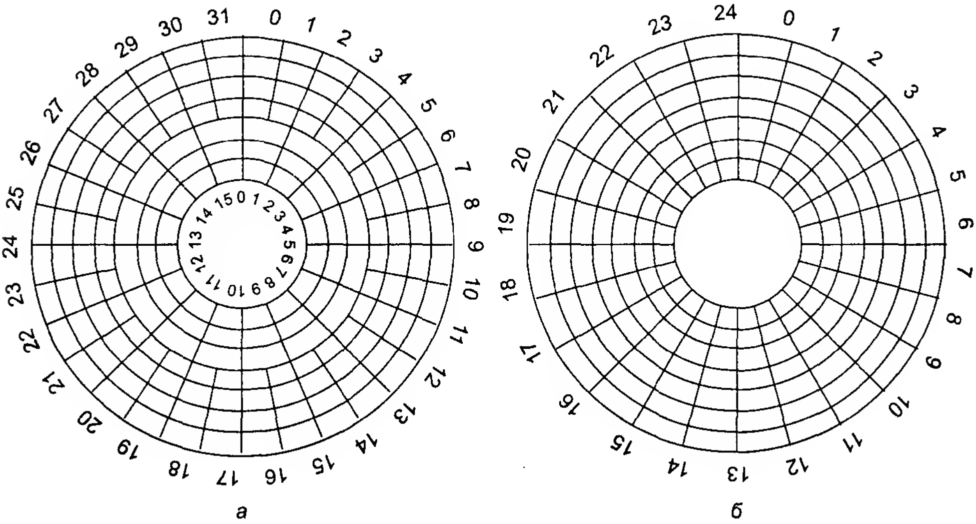


Рис. 5.14. Физическая геометрия диска с двумя зонами (*а*); возможная виртуальная геометрия этого диска (*б*)

Чтобы не усложнять жизнь операционной системы излишними подробностями, современные жесткие диски скрывают истинную геометрию и предоставляют в качестве интерфейса виртуальную геометрию с одинаковым числом секторов на всех цилиндрах. Встроенный микроконтроллер диска преобразует обращение к виртуальным цилиндру, головке и сектору в физические соответствующие параметры. Возможная виртуальная геометрия диска, изображенного на рис. 5.14, *а*, показана на рис. 5.14, *б*. В обоих случаях диск имеет 192 сектора.

Для компьютеров с процессором Pentium максимальными значениями этих трех параметров часто являются значения 65 535, 16 и 63, что вызвано необходимостью обратной совместимости с ограничениями оригинальной машины IBM PC. На ней для хранения этих значений использовались 16-, 4- и 6-разрядные двоичные поля, причем номера цилиндров и секторов начинались с 1, а номера головок — с 0. При таких параметрах и 512 байтах на сектор максимальный размер поддерживаемого диска равен 31,5 Гбайт. Чтобы преодолеть это ограничение, многими дисками теперь поддерживается режим **LBA** (Logical Block Addressing — логическая адресация блоков), при котором секторы диска просто нумеруются последовательно начиная с 0, независимо от геометрии диска.

RAID

Производительность центральных процессоров за последнее десятилетие увеличивалась экспоненциально, удваиваясь примерно каждые 18 месяцев. С производительностью дисков дело обстояло совсем не так. В 70-е годы среднее время поиска на дисках, используемых в мини-компьютерах, составляло от 50 до 100 мс. Сегодня время поиска немного ниже 10 мс. В большинстве отраслей промышленности (например, в автомобильной или авиационной) увеличение производи-

тельности в 5 или 10 раз за два десятилетия считалось бы феноменальным, но в компьютерной промышленности такой низкий рост является препятствием. За эти годы разрыв между производительностью центрального процессора и производительностью диска только еще более увеличился, причем весьма существенно.

Как мы видели, для увеличения производительности центрального процессора все больше используются параллельные вычисления. На протяжении десятилетий идея распараллелить операции ввода-вывода также приходила в головы многих людей. В 1988 году в своей статье Паттерсон и его коллеги предложили шесть различных способов организации дисков для улучшения производительности или надежности дисковых операций, либо и того и другого [261]. Эти идеи были быстро приняты промышленностью, в результате чего был разработан новый класс устройств ввода-вывода, названный **RAID**. Паттерсон с соавторами определили RAID как Redundant Array of Inexpensive Disks — массив недорогих дисков с избыточностью, но промышленники переопределили букву I как «Independent» (независимые). Возможно, это должно было им позволить продавать диски по высоким ценам. Поскольку в этой пьесе также требовался кто-нибудь на роль злодея (как и в случае противостояния RISC и CISC, также благодаря Паттерсону), «негодяя» назвали **SLED** (Single Large Expensive Disk — одиночный большой и дорогой дисковый накопитель).

Идея, лежащая в основе системы RAID, состоит в том, что на компьютер (обычно большой сервер) устанавливается коробка, полная дисков. Вместо обычного дискового контролера устанавливается специальный RAID-контроллер, а весь набор дисков выглядит с точки зрения операционной системы как один большой (и дорогой) дисковый накопитель, то есть SLED, но обладает более высокой производительностью и большей надежностью. Поскольку SCSI-диски отличаются высокой производительностью, низкой ценой и способны работать до 7 штук на одном контроллере (до 15 для так называемого «широкого» SCSI), неудивительно, что большинство RAID-систем состоят из RAID-контроллера SCSI и коробки SCSI-дисков. Операционная система воспринимает их как один большой диск. Таким образом, для использования системы RAID не требуется никаких изменений программного обеспечения, что является большим плюсом с точки зрения системных администраторов.

Еще одно свойство всех RAID-систем состоит в том, что данные распределяются по дискам, а это позволяет распараллеливать операции. Паттерсоном и его коллегами было предложено несколько различных схем использования системы RAID, получивших известность как уровни от нулевого до пятого. Помимо них существует еще несколько второстепенных уровней, которые мы не станем здесь обсуждать. Употребление термина «уровень» не совсем правильно в этом случае, так как данные схемы использования системы RAID не образуют иерархии. Есть просто шесть различных вариантов организации совместной работы дисков.

Система RAID уровня 0 проиллюстрирована на рис. 5.15, *а*. Она рассматривает единый виртуальный диск, эмулируемый контроллером RAID, как разбитый на полосы, состоящие из одинакового числа секторов (обычно по 64 Кбайт), пересекающие наборы дисков так, что первый блок секторов записывается на первый диск, второй блок — на второй диск и т. д. по кругу. На рис. 5.15, *а* показана система RAID уровня 0 для четырех дисков. Подобный способ хранения данных

на нескольких дисках называется **чередующимся набором**. При этом, если программа издает запрос чтения или записи целой полосы данных, RAID-контроллер разбивает этот запрос на отдельные запросы по числу дисков и адресует каждый запрос отдельному диску. Таким образом, все диски системы RAID работают параллельно, причем программное обеспечение об этом может даже не догадываться. Восстановление вышедшего из строя диска в этом случае будет значительно сложнее, чем в системе RAID уровня 4[1].

Система RAID уровня 0 лучше всего работает с большими запросами, что естественно, так как при этом работа более равномерно распределяется между дисками. В результате производительность такой системы оказывается достаточно высокой при довольно простой реализации.

Хуже всего система RAID уровня 0 работает с операционными системами, имеющими привычку запрашивать данные по одному сектору. Запрос будет выполнен верно, но параллельной загрузки дисков при этом не получится и, соответственно, не будет и выигрыша в производительности. Еще один недостаток системы RAID уровня 0 заключается в том, что надежность такой системы потенциально ниже, чем у одного диска (SLED). Так, если система RAID состоит из четырех дисков со средней наработкой на отказ для каждого диска, равной 20 000 часов, то при совместном использовании этих дисков примерно раз в 5000 часов один из дисков будет отказывать, что приведет к отказу всей системы RAID. Причем в этом случае могут быть потеряны все данные. Таким образом, получается, что SLED оказывается в четыре раза надежнее. Поскольку в системе RAID уровня 0 избыточность не предусмотрена, она не считается настоящей RAID-системой.

Следующий вариант, RAID уровня 1, показанный на рис. 5.15, *б*, представляет собой настоящую систему RAID. В ней дублируются все диски, в результате имеются четыре основных и четыре резервных диска. Такая система RAID называется также зеркальным набором. При записи каждая полоса записывается дважды. При чтении может использоваться любая копия. Таким образом, по сравнению с системой RAID уровня 0 скорость записи не увеличивается, а чтение может быть ускорено вдвое. Отказоустойчивость такой системы очень хороша, так как все данные дублируются полностью. Для восстановления системы при выходе из строя одного из дисков нужно всего лишь заменить диск и скопировать на него данные с диска-копии. Недостатком является снижение используемой общей емкости дисков вдвое.

В отличие от уровней 0 и 1, работающих с полосами и секторами, система RAID уровня 2 работает на уровне слов и даже байтов. Представьте разбиение каждого байта единого виртуального диска на пару 4-битовых полубайтов, затем добавление к каждому из них кода Хэмминга с образованием 7-битового слова, в котором биты 1, 2 и 4 являются битами четности. Затем представьте, что семь дисков на рис. 5.15, *в* синхронизированы по скорости вращения и позиции головок. В этом случае будет возможно записать закодированное по Хэммингу 7-битовое слово на семь дисков, по биту на диск.

---

[1] Точнее, процесс восстановления абсолютно ничем не отличается от предыдущего случая. В обоих случаях нужно всего лишь сложить все данные на оставшихся дисках по модулю 2, что обусловливается природой этой арифметической операции, являющейся обратной для самой себя. Немного сложнее для системы будет лишь процесс обычной записи на RAID уровня 5. — *Примеч. перев.*

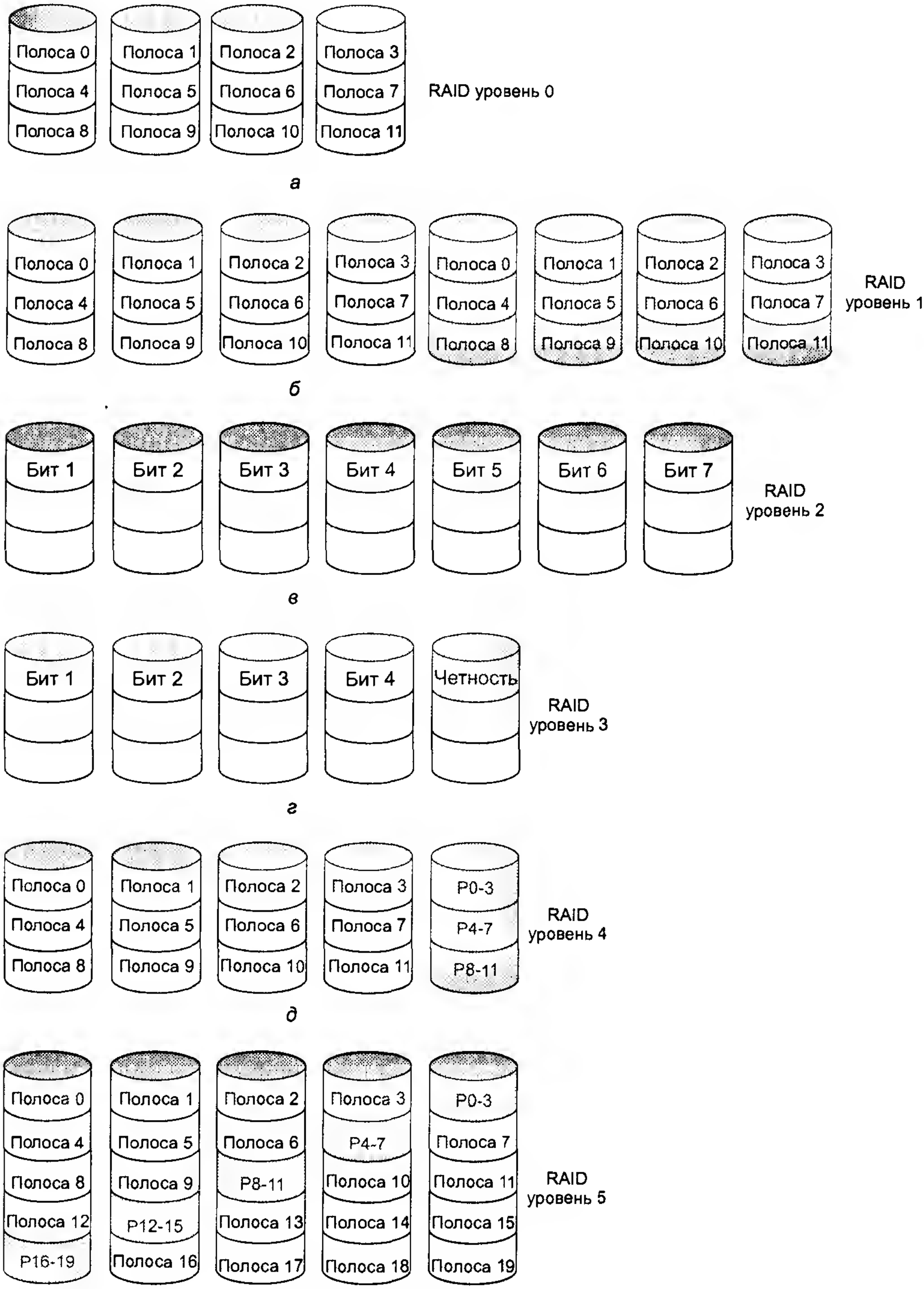


Рис. 5.15. Системы RAID уровней от 0 до 5. Резервные диски затенены

Эта схема использовалась в компьютере CM-2 фирмы Thinking Machines. Бралось 32-разрядное слово данных, к нему добавлялись 6 битов четности, чтобы образовать 38-разрядное слово Хэмминга, плюс дополнительный бит четности. Полученное 39-разрядное слово записывалось на 39 дисков. Таким образом, скорость операций чтения и записи увеличивалась в 32 раза. Потеря одного из устройств также не вызывала особых проблем, поскольку это приводило к потере всего одного бита в 39-разрядном слове, с чем код Хэмминга легко справлялся на лету.

Недостаток этой схемы заключался в том, что для ее работы требовалась синхронизация вращения всех дисков. Кроме того, такая схема имела смысл только при значительном количестве дисков. Даже при 32 дисков с данными и 6 с битами четности накладные расходы составляли 19 %. Кроме того, контроллер должен постоянно и с большой скоростью считать контрольную сумму по Хэммингу.

Система RAID уровня 3 представляет собой упрощенную версию системы RAID уровня 2. Одна показана на рис. 5.15, *г*. В этой схеме для каждого слова данных считается один бит четности, записываемый на отдельный диск четности. Как и в случае системы RAID уровня 2, все диски должны быть точно синхронизированы, так как слово данных побитно пишется сразу на все диски.

На первый взгляд может показаться, что одиночный бит четности дает только возможность выявления ошибок, но не их исправления. Для случайных ошибок в произвольном разряде это справедливо. Однако для случая выхода из строя диска одного бита вполне достаточно для полного восстановления данных, так как позиция бита известна. В случае выхода диска из строя контроллер просто считает, что все биты, содержащиеся на нем, равны нулю, определяя их истинное значение по четности разрядов, считанных с остальных дисков. Хотя оба уровня 2 и 3 системы RAID позволяют добиться очень высоких скоростей передачи данных, число отдельных запросов ввода-вывода, которые они могут обработать, не выше, чем у отдельного диска.

Системы RAID уровней 4 и 5 снова работают с полосами (чередующимися наборами данных), а не с отдельными словами с битами четности, поэтому не требуют синхронизации дисков. Система RAID уровня 4 (рис. 5.15, *д*) аналогична уровню 0, но с дополнительным диском четности, содержащим сумму по модулю два всех данных с остальных дисков. Если любой из дисков выйдет из строя, потерянные байты могут быть восстановлены при помощи той же операции сложения по модулю два.

Такая организация надежно защищает от выхода диска из строя, но при небольших изменениях информации, хранящейся на дисках, производительность не только не увеличивается, а даже наоборот, снижается. Если изменяется всего один сектор, то все равно необходимо прочитать данные со всех дисков, чтобы заново рассчитать значения битов четности, которые нужно будет перезаписать. В качестве альтернативы можно читать старые данные пользователя и старое значение избыточной информации и пересчитывать новое значение по этим данным. Но и такой вариант все равно требует дополнительных операций чтения перед операциями записи.

В результате повышенной нагрузки на диск, содержащей избыточные данные, этот диск может стать узким местом системы. Эта проблема решается в системе RAID уровня 5, в которой биты четности равномерно распределены по всем дискам, как показано на рис. 5.15, *е*. На первый взгляд может показаться, что восстановле-

ние вышедшего из строя диска в этом случае будет значительно сложнее, чем в системе RAID уровня 4. Однако на самом деле оно ничем не отличается от предыдущего случая. В обоих случаях нужно всего лишь сложить все данные на оставшихся дисках по модулю 2, что обусловливается природой этой арифметической операции, являющейся обратной для самой себя.

Компакт-диски

В 70-е годы появились оптические диски (в противоположность магнитным). Их плотность записи сразу значительно превзошла плотность записи магнитных дисков. Изначально оптические диски были разработаны для записи телевизионных программ, но затем для них были придуманы другие сферы применения, в том числе хранение компьютерных данных. Благодаря их потенциально огромной емкости, оптические диски стали предметом большого числа исследований и совершили невероятно быструю эволюцию.

Оптические диски первого поколения были разработаны голландским электронно-промышленным конгломератом Philips для хранения фильмов. Они были 30 см в диаметре и продавались на рынке под названием LaserVision, но без особого коммерческого успеха, разве что в Японии.

В 1980 году фирма Philips совместно с Sony разработала компакт-диск (CD, Compact Disc), который быстро вытеснил вращающиеся со скоростью 33 1/3 оборотов в минуту виниловые пластинки (хотя до сих пор отдельные люди, называющие себя «аудиофилами», утверждают, что качество звука у винилов выше, чем у лазерных компакт-дисков). Точные технические детали компакт-диска были опубликованы в виде Международного стандарта IS 10149, называемого в народе **Красной книгой** из-за цвета обложки. Международные стандарты издаются Международной организацией по стандартизации ISO (International Organization for Standardization). Каждому международному стандарту обычно соответствует какой-либо национальный стандарт, как, например, ANSI (American National Standards Institute — Национальный институт стандартизации США) или DIN (Deutsche Industrie Norm — промышленная норма Германии), применяемые в Европе. Публикация спецификаций компакт-диска в качестве международного стандарта обеспечивает совместную работу компакт-дисков и проигрывателей для них, выпущенных производителями всего мира. Все компакт-диски должны иметь диаметр 120 мм и толщину 1,2 мм, с 15-мм отверстием в середине. Аудиокомпакт-диски стали первым цифровым носителем, завоевавшим успех на массовом рынке. Предполагается, что они будут сохраняться до 100 лет. Пожалуйста, проверьте в 2080 году, так ли это.

Для производства компакт-дисков используется мощный инфракрасный лазер, которым прожигаются отверстия диаметром 0,8 мкм в металлическом покрытии стеклянного диска-оригинала, называемого также мастер-диском. С мастер-диска снимается слепок с выпуклостями на месте прожженных лазером отверстий. С помощью этого слепка из поликарбонатной смолы печатаются компакт-диски с тем же рисунком выемок, что и на стеклянном оригинале. Затем застывший поликарбонат покрывается очень тонким слоем отражающего свет алюминия, поверх которого диск покрывается защитным лаком, а на него затем наносится этикетка. Углубления в поликарбонате называют **питами** (pit — впадина), а нетронутые лазером участки между питами называют «землей» или «сушей» (land).

При воспроизведении полупроводниковый лазер низкой мощности освещает питы и участки между ними лучом с длиной волны 0,78 мкм, в то время как они прокручиваются с постоянной линейной скоростью. Лазер освещает алюминиевое покрытие диска сквозь прозрачный поликарбонатный диск толщиной в 1,2 мм. Питы выглядят с этой стороны как выступы высотой в четверть длины волны луча лазера. В результате интерференции свет, отраженный от питов, имеет меньшую яркость, чем отраженный от участков между ними, что и регистрируется фотодетектором как последовательность нулей и единиц. На самом деле для большей надежности за 1 считается переход пит/земля или земля/пит, а отсутствие перехода означает 0.

Последовательность питов и промежутков между ними записывается в виде единой непрерывной спиральной дорожки, начинающейся недалеко от центра диска и заканчивающейся у края диска. Максимальная ширина спирали может составлять 32 мм, а число оборотов — 22 188 (около 600 на мм). Длина спирали, показанной на рис. 5.16, достигает 5,6 км в длину.

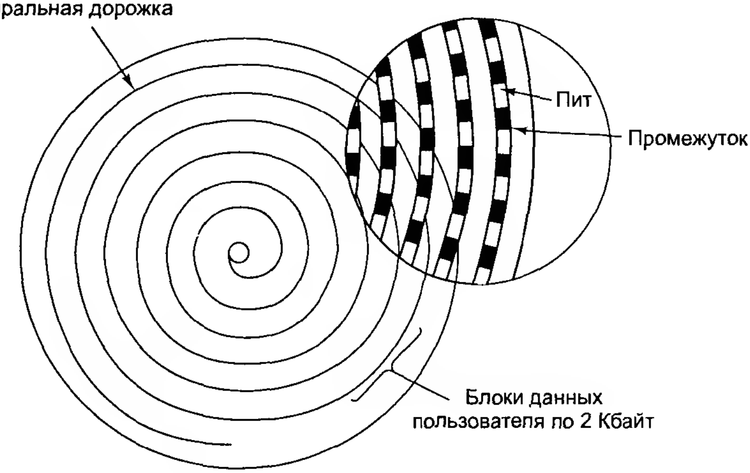


Рис. 5.16. Структура записи компакт-диска или CD-ROM

Чтобы музыка воспроизводилась с постоянной скоростью, поток питов и промежутков между ними должен двигаться под лучом лазера с постоянной линейной скоростью в 120 см/с. Соответственно, по мере продвижения считывающей головки от центра диска к краю, угловая скорость вращения компакт-диска должна постоянно уменьшаться от 530 об/мин до 200 об/мин. В этом принципиальное отличие оптических дисков от магнитных, вращающихся с постоянной угловой скоростью, не зависящей от текущего положения головок[1]. Кроме того, 530 об/мин — это существенно меньше, чем 3600 об/мин или 7200 об/мин, типичная скорость вращения магнитных дисков.

В 1984 году фирмы Philips и Sony осознали потенциал использования компакт-дисков для хранения компьютерных данных, поэтому они опубликовали **Желтую**

[1] Требование постоянной линейной скорости значительно усложняет механизм и требует постоянных затрат энергии и времени на разгон и торможение диска. — *Примеч. перев.*

книгу, в которой определили точный стандарт того, что теперь называется **CD-ROM** (Compact Disk Read Only Memory — компакт-дисковое постоянное запоминающее устройство). Поскольку ранок аудиокомпакт-дисков к тому моменту был уже весьма широк, было решено сделать CD-ROM точно того же размера, что и аудиокомпакт-диски, а также механически и оптически совместимыми с ними. Помимо возможности проигрывать музыку на устройствах чтения CD-ROM, это также давало возможность использовать для производства CD-ROM те же технологические линии, на которых производились аудиокомпакт-диски. В результате себестоимость производства одного CD-ROM оказалась намного ниже одного доллара. Недостатком такого решения явилась необходимость использования в устройствах чтения CD-ROM медленных двигателей, вращающихся с переменной скоростью.

В Желтой книге был определен формат компьютерных данных. Новый стандарт также улучшал способность системы исправлять ошибки, что было очень важно, так как если на слух почти не заметно искажение одного бита то тут то там, для хранения компьютерной информации требуется значительно более высокая надежность. Основой формата CD-ROM является кодирование одного байта 14-разрядным числом. Как уже было сказано, 14 разрядов достаточно для кодирования одного байта (8 бит) кодом Хэмминга с запасом в два бита. В действительности используется более мощная система помехоустойчивого кодирования. При чтении преобразование 14-в-8 осуществляется аппаратно при помощи таблицы.

На более высоком уровне группа из 42 последовательных 14-разрядных символов образуют 588-разрядный **кадр**. Каждый кадр содержит 192 бит данных (24 байт). Оставшиеся 396 бит используются для коррекции ошибок и управления. До сих пор используемая схема одинакова для звуковых компакт-дисков и CD-ROM.

Желтая книга добавляет к этому стандарту группирование 98 кадров в так называемый **CD-ROM-сектор**, как показано на рис. 5.17. Каждый CD-ROM-сектор начинается 16-байтовым заголовком, первые 12 байт которого содержат 00FFFFFFFFFFFFFFFFFFFF00 (шестнадцатеричное), чтобы считывающее устройство могло распознать начало CD-ROM-сектора. Следующие три байта содержат номер сектора, что необходимо, так как поиск данных на единственной спирали диска значительно сложнее, чем на магнитном диске, состоящем из концентрических дорожек. При поиске программное обеспечение устройства приблизительно вычисляет, куда переместить головку, а после перемещения головки считывает первый заголовок, проверяя, насколько точно удалось приблизиться к требуемым данным. Последний байт заголовка содержит код режима.

Желтой книгой определяется два режима. В режиме 1 используется формат, показанный на рис. 5.17, с 16-байтовым заголовком, 2048 байт данных и 288-байтовым кодом исправления ошибок ECC (Error Correction Code), для которого используется перемежающийся код Рида—Соломона. В режиме 2 поле данных объединяется с полем ECC в 2336-байтное поле данных. Этот режим может использоваться для тех приложений, которым не требуется (или у них нет на это времени) коррекция ошибок, например аудио или видео. Обратите внимание, что коррекция ошибок осуществляется на трех уровнях: внутри символа, в кадре и в CD-ROM-секторе. Однобитовые ошибки исправляются на нижнем уровне, короткие

пакеты ошибок корректируются на уровне кадров, а все остальные ловятся на уровне сектора. В результате 98 кадров по 588 бит (7203 байт) содержат всего лишь 2048 байт полезной нагрузки, что соответствует эффективности около 28 %. Такую цену приходится платить за надежность хранения информации на оптическом диске.

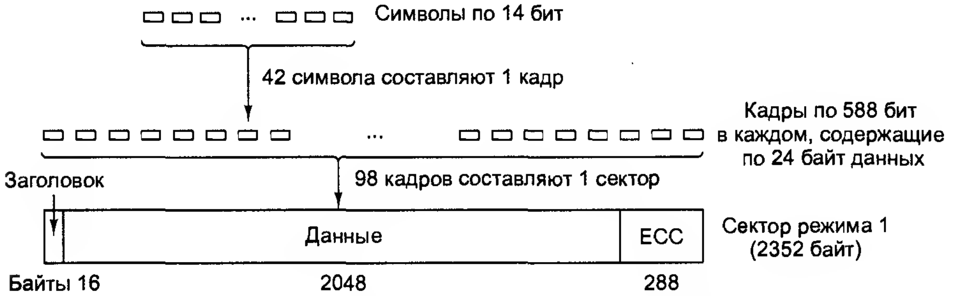


Рис. 5.17. Логическое расположение данных на CD-ROM

Односкоростные CD-ROM-приводы считывают данные со скоростью 75 секторов в секунду, что соответствует скорости передачи данных 153 600 байт/с в режиме 1 и 175 200 байт/с в режиме 2. Двухскоростные приводы в два раза быстрее и т. д. Таким образом, 40-скоростной CD-ROM-привод может поставлять данные со скоростью до 40×153 600 байт/с (6000 Кбайт/с), при условии, что интерфейс устройства, шина и операционная система могут обработать такой поток данных. На стандартном аудиокомпакт-диске помещается до 74 мин музыки, что при использовании его для хранения данных в режиме 1 дает емкость в 681 984 000 байт. Эта величина обычно обозначается как 650 Мбайт, так как 1 Мбайт равен 2^{20} байт (1 048 576 байт), а не 1 000 000 байт.

Обратите внимание, что даже 32-скоростное устройство чтения CD-ROM (4 915 200 байт/с) не ровня магнитному диску SCSI-2 со скоростью в 10 Мбайт/с, несмотря на то что во многих CD-ROM-приводах используется интерфейс SCSI (IDE CD-ROM-приводы также существуют). Если задуматься о времени поиска для устройств чтения CD-ROM, составляющем несколько сот миллисекунд, станет ясно, что по производительности эти устройства находятся совсем не в той «весовой» категории, где жесткие магнитные диски, несмотря на довольно внушительную емкость.

В 1986 году фирма Philips снова нанесла удар, выпустив **Зеленую книгу**, добавив к стандарту графику и возможность совмещения в одном секторе аудио, видео и данных, что существенно для мультимедийных компакт-дисков.

Наконец, следует рассмотреть файловую систему, используемую на CD-ROM. Чтобы один и тот же CD-ROM можно было использовать на различных компьютерах, необходимо добиться соглашения по вопросу файловых систем для CD-ROM. Для достижения этого соглашения представители многих компьютерных компаний встретились на озере Тахо, в Хай Сьеррас, на границе Калифорнии и Невады. По названию местности была названа разработанная ими файловая система **High Sierra**. Позднее она была оформлена в виде Международного стандарта IS 9660.

Файловая система High Sierra состоит из трех уровней. На первом уровне файлы имеют имена формата, схожего с MS-DOS — до 8 символов имя файла плюс до 3 символов расширение. В имени файла могут использоваться только прописные символы, цифры и символ подчеркивания. Глубина вложенности каталогов ограничена восемью. Имена каталогов не могут содержать расширений. Все файлы уровня 1 должны быть непрерывными, чего не сложно добиться на носителе, на который информация записывается всего один раз. Таким образом, любой CD-ROM, соответствующий уровню 1 стандарта IS 9660, может быть прочитан в системе MS-DOS, на компьютере Apple, в системе UNIX, то есть практически на любом компьютере. Производители CD-ROM считают это свойство большим плюсом.

Уровень 2 стандарта IS 9660 разрешает использовать имена файлов длиной до 32 символов, а уровень 3 позволяет использовать сегментированные файлы. Расширения стандарта Rock Ridge (прихотливо названное в честь города в фильме Джина Уайлдера *Blazing Saddles*) разрешают использовать очень длинные имена (для UNIX), а также универсальные идентификаторы UID, глобальные идентификаторы GID и связывание файлов. Однако CD-ROM, не удовлетворяющие уровню 1 стандарта IS 9660, могут оказаться нечитаемыми на многих компьютерах.

CD-ROM становятся все более популярными для распространения игр, фильмов, энциклопедий, атласов, учебников и т. п. Большая часть коммерческого программного обеспечения сегодня распространяется на CD-ROM. Благодаря сочетанию большой емкости и низкой цены CD-ROM удачно подходят для самых разнообразных приложений.

Компакт-диски с возможностью записи

Сначала оборудование, необходимое для создания CD-ROM (или аудиокомпакт-диска), было крайне дорогим. Но как это обычно бывает в компьютерной промышленности, ничто не остается дорогим надолго. К середине 90-х годов устройства для записи компакт-дисков размером не более проигрывателей компакт-дисков стали обычными периферийными устройствами, продающимися в большинстве компьютерных магазинов. Эти устройства по-прежнему отличались от магнитных дисков, так как позволяли только один раз записывать данные, которые потом было нельзя стереть. Тем не менее они быстро нашли свою нишу в качестве носителей для резервных копий больших жестких дисков, а также позволили отдельным пользователям или небольшим компаниям производить свои небольшие серии CD-ROM или создавать оригиналы-макеты для доставки их на коммерческие фабрики по производству компакт-дисков. Эти носители известны под названием **CD-R** (CD-Recordable — компакт-диски с возможностью записи).

Физически компакт-диски с возможностью записи, так же как и обычные CD-ROM, состоят из 120-мм поликарбонатных пластин, с той разницей, что на них нанесена спиральная дорожка глубиной 0,6 мм для направления луча лазера при записи. Дорожка выполнена в виде синусоиды с амплитудой в 0,3 мм и частотой 22,05 кГц (при одинарной скорости вращения диска), что обеспечивает постоянную обратную связь, позволяющую отслеживать и корректировать скорость вращения диска. CD-R выглядят как обычные CD-ROM, отличаясь от них в основ-

ном этикеткой и цветом. С верхней стороны CD-R золотые, в отличие от серебристых обычных компакт-дисков. Этот цвет вас не обманывает. Это действительно тонкий слой золота, используемый для отражения луча лазера вместо алюминия. В отличие от обычных серебристых компакт-дисков, на поверхности которых пресс-формой нанесены углубления, на CD-R-дисках отличия в отражающей способности питов и промежутков между ними должны имитироваться другим способом. Это достигается при помощи добавления специального слоя красителя между поликарбонатом и отражающей золотой поверхностью, как показано на рис. 5.18. Используются два типа красителя: синевато-зеленый цианин и желтовато-оранжевый фталоцианин. Химики могут спорить до хрипоты, доказывая преимущества одного красителя над другим. Эти красители сходны с используемыми в фотографии, что объясняет, почему корпорации Eastman Kodak и Fuji являются основными производителями чистых CD-R.

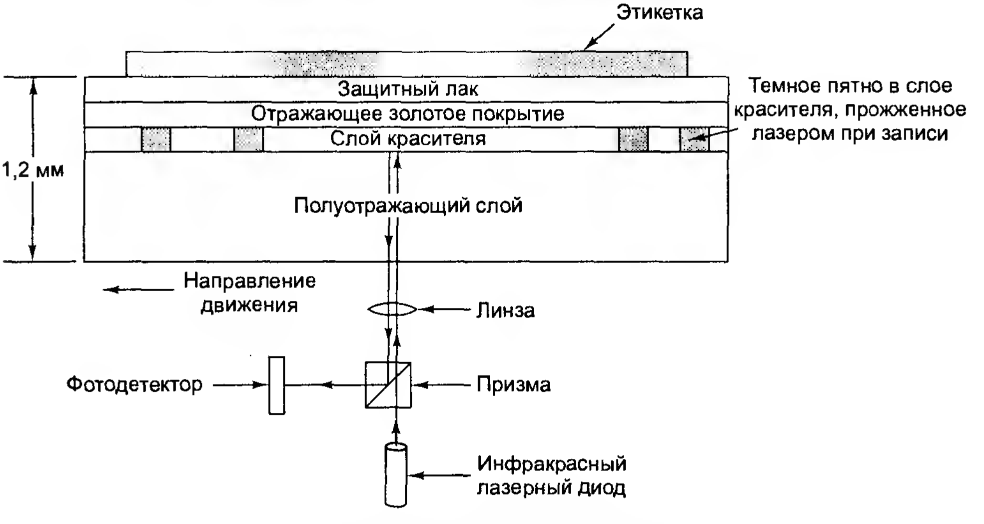


Рис. 5.18. Поперечный разрез CD-R и лазера (не в масштабе). Серебристые CD-ROM имеют сходную структуру с той разницей, что у них нет слоя красителя, а вместо золотого покрытия — алюминиевое с углублениями

В изначальном состоянии уровень красителя прозрачен и позволяет лучу лазера проходить сквозь него и отражаться от золотого покрытия. Во время записи лазер переводится в режим высокой мощности (8–16 мВт). Когда луч лазера попадает на краситель, он нагревает его, разрушая химические связи. При этом образуется темное пятно. При чтении лучом лазера с мощностью 0,5 мВт фотодетектор замечает разницу между темными пятнами, прожженными лазером, и нетронутыми прозрачными областями. Это различие интерпретируется так же, как и разница между выемками и ровной поверхностью у обычных компакт-дисков, даже на обычных устройствах чтения CD-ROM и проигрывателях аудиокомпакт-дисков.

Ни одна новая разновидность компакт-дисков не может обойтись без книги соответствующего цвета, поэтому в 1989 году была опубликована **Оранжевая книга**, посвященная компакт-дискам с возможностью записи. Этот документ опре-

деляет формат CD-R, а также новый формат, **CD-ROM XA**, позволяющий посекторно дописывать информацию на CD-R. Группа последовательных секторов, записываемых за один раз, называется **CD-ROM-дорожкой**.

Одним из первых применений CD-R была система Kodak PhotoCD, в которой диск использовался в качестве фотоальбома. Клиент приносил отснятую фотопленку и свой старый PhotoCD в киоск фирмы Kodak и получал обратно тот же самый диск с дописанными на него новыми фотографиями. Новый пакет фотографий, созданных при сканировании негативов, записывается на PhotoCD в виде отдельной CD-ROM-дорожки. Инкрементная запись требовалась, так как в момент появления первых CD-R на рынке их цена была слишком высока, чтобы использовать в качестве одноразового носителя.

Однако инкрементная запись создает новую проблему. До выхода в свет Оранжевой книги у всех CD-ROM было единое оглавление тома **VTOC** (Volume Table of Contents) в начале диска. При использовании инкрементной (многодорожечной) записи такая схема перестает работать. Решение, опубликованное в Оранжевой книге, состоит в том, что для каждой CD-ROM-дорожки создается собственное оглавление тома VTOC. В нем могут перечисляться некоторые или все файлы предыдущих дорожек. Когда CD-R вставляется в дисковод, операционная система перебирает все CD-ROM-дорожки в поисках самого последнего оглавления тома, соответствующего текущему состоянию диска. Включением в новое оглавление тома не всех файлов из предыдущего оглавления создается иллюзия удаления файлов с диска. Дорожки могут объединяться в **сеансы**, образуя диски с **многосеансовой** записью. Стандартные проигрыватели компакт-дисков не поддерживают многосеансовые компакт-диски, так как они рассчитаны на единственное оглавление тома в начале диска.

Каждая дорожка должна быть записана на диск за одну непрерывную (без остановок) операцию. В результате жесткий диск, с которого производится запись, должен быть достаточно быстрым, чтобы успевать предоставлять записываемые данные вовремя. Если копируемые на диск файлы разбросаны по всему жесткому диску, время поиска может оказаться слишком долгим, что приведет к опустошению буфера и «пересыханию» потока данных, поступающих на CD-R. В результате вместо диска с записями вы получите блестящую (но несколько дороговатую) подставку под стаканы для прохладительных (или горячительных) напитков, которая также неплохо летает. Программное обеспечение для устройств записи CD-R обычно предлагает собрать все записываемые файлы в виде единого непрерывного образа CD-ROM, размером в 650 Мбайт, прежде чем прожигать CD-R, но эта процедура удваивает общее время записи диска и требует 650 Мбайт свободного места на жестком диске. К тому же даже такая мера не спасает от случайностей, как, например, перегрев жесткого диска, в результате которого он вдруг начинает выполнять операцию термальной калибровки.

Компакт-диски с возможностью записи упрощают отдельным пользователям и компаниям задачу перезаписи CD-ROM и аудиокомпакт-дисков, обычно нарушая авторские права издателя оригинального диска. Для усложнения жизни пиратов были разработаны различные схемы, в частности усложняющие чтение CD-ROM чем-либо, не являющимся программным обеспечением издателя. Одна из таких схем заключается в указывании фиктивной многогигабайтной длины файлов, что

усложняет копирование файлов на жесткий диск при помощи стандартных программных средств. Истинные длины файлов упрятаны в программном обеспечении издателя или в неожиданном месте компакт-диска, возможно, в зашифрованном виде. Другая схема заключается в использовании намеренно неверной ECC-информации в некоторых секторах в расчете на то, что программное обеспечение, копирующее компакт-диск, «исправит» эти ошибки. Программное обеспечение издателя само проверяет коды исправления ошибок, и если они верны, то отказывается работать. Также возможно использование нестандартных промежутков между дорожками и других физических «дефектов».

Многократно перезаписываемые компакт-диски

Хотя люди привыкли пользоваться носителями, позволяющими однократную запись, такими как бумага или фотопленка, спрос на CD-ROM с возможностью многократной перезаписи оставался. Такая технология недавно появилась под названием **CD-RW** (CD-ReWritable — многократно перезаписываемые компакт-диски). CD-RW по размерам ничем не отличаются от обычных компакт-дисков, CD-ROM или CD-R. Однако вместо цианина или фталоцианина в CD-RW в качестве записывающего слоя используется сплав серебра, индия, сурьмы и теллура. У этого сплава два устойчивых состояния: кристаллический и аморфный, с различной отражающей способностью.

В устройствах записи CD-RW используются лазеры с тремя различными уровнями мощности. При высокой мощности лазер расплавляет сплав, преобразуя его из кристаллического состояния с высокой отражающей способностью в аморфное состояние с низкой отражающей способностью, соответствующее питу. При средней мощности лазер плавит сплав, превращая его в кристаллическое состояние, соответствующее промежуткам между питами. При низкой мощности лазер считывает информацию с диска, не вызывая фазовых переходов сплава.

Причина, по которой CD-RW не вытеснили с рынка CD-R, заключается в их значительно более высокой цене. Кроме того, для архивирования жестких дисков невозможность случайно стереть CD-R является большим плюсом.

DVD

Базовый формат компакт-дисков (CD/CD-ROM) не менялся с 1980 года. Однако технология с тех пор ушла вперед, в результате чего стало экономически возможным создание оптических дисков большей емкости. Спрос на такие диски уже весьма велик и продолжает возрастать. Голливуд хотел бы заменить видеоленты цифровыми дисками, поскольку диски обеспечивают более высокое качество изображения, дешевле в производстве, долговечнее, занимают меньше места на полках магазинов и не требуют перемотки. Компании, занимающиеся производством электронных потребительских товаров, ищут новый продукт, который бы вызвал очередной покупательский бум, а различные компьютерные компании мечтают добавить к своему программному обеспечению черты мультимедиа.

В результате объединения технологических усилий и интересов трех богатейших и мощнейших отраслей промышленности и была создана система **DVD**. Сокращение DVD, изначально расшифровывающееся как Digital Video Disk (циф-

ровой видеодиск), теперь официально обозначает Digital Versatile Disk (универсальный цифровой диск). В DVD используется та же самая общая схема, что и для компакт-дисков, то есть те же 120-мм поликарбонатные диски, изготавливаемые печатным способом, содержащие углубления, освещаемые лазером и считываемые фотодетектором. Новым для DVD было:

1. Вдвое меньший размер питов (0,4 мкм против 0,8 мкм для компакт-дисков).
2. Более тугая спираль (0,74 микрона между соседними дорожками, вместо 1,6 мкм для компакт-дисков).
3. Красный лазер (с длиной волны 0,65 мкм вместо 0,78 мкм для компакт-дисков).

Вместе эти усовершенствования позволили увеличить емкость в семь раз, то есть с 650 Мбайт до 4,7 Гбайт. Односкоростное устройство чтения DVD считывает данные со скоростью 1,4 Мбайт/с (против 150 Кбайт/с для компакт-дисков). К сожалению, переход на красный лазер (вроде тех, что используются в супермаркетах) означает необходимость установки второго лазера или сложной преобразовательной оптики для возможности считывать на новом устройстве обычных компакт-дисков. Такую возможность обеспечивают далеко не все устройства чтения DVD. Кроме того, чтение CD-R и CD-RW на DVD-проигрывателе может также оказаться невозможным.

Достаточно ли емкости 4,7 Гбайт? Возможно. При использовании алгоритма сжатия MPEG-2 (Международный стандарт IS 13346) диск емкостью 4,7 Гбайт может вместить до 133 мин полноэкранного динамичного видеоизображения высокого разрешения (720×480), а также звуковую дорожку на восьми языках плюс субтитры еще на 32 языках. Около 92 % всех фильмов, сделанных в Голливуде, короче 133 мин. Тем не менее для некоторых приложений, таких как мультимедиа, игры или энциклопедии, может понадобиться носитель большего размера, кроме того, возможно, кому-нибудь захочется поместить на диск более одного фильма. Поэтому были определены четыре следующих формата:

1. Односторонний, одноуровневый (4,7 Гбайт).
2. Односторонний, двухуровневый (8,5 Гбайт).
3. Двусторонний, одноуровневый (9,4 Гбайт).
4. Двусторонний, двухуровневый (17 Гбайт).

Зачем так много форматов? Ответ: политика. Фирмы Philips и Sony хотели создать односторонний, двухуровневый диск, но фирмам Toshiba и Time Warner был нужен двусторонний, одноуровневый. Фирмы Philips и Sony полагали, что клиенты вряд ли будут рады необходимости переворачивать диск, а компания Time Warner не верила, что двусторонние диски будут работать. В результате было принято компромиссное решение: принять стандарты для всех четырех возможных комбинаций, а рынок пусть сам разберется, какой из этих стандартов выживет.

Двухуровневая технология заключается в том, что у диска есть с краю отражающий слой и полуотражающий слой посредине. Считываться будет тот уровень, на который будет сфокусирован лазер. Для нижнего уровня требуются питы и промежутки большего размера, чтобы надежность считывания не пострадала. В результате нижний уровень оказался немного менее емким.

Двусторонний диск представляет собой просто два обычных 0,6-мм диска, склеенных спинками. Чтобы толщина всех типов дисков была одинаковой, односто-

ронний диск состоит из обычного диска толщиной 0,6 мм, приклеенного к пустому поликарбонатному основанию. (Возможно, в будущем вместо пустышки будут приклеивать 133 мин рекламы, в надежде, что покупатели из любопытства заглянут туда.) Структура двустороннего двухуровневого диска показана на рис. 5.19.

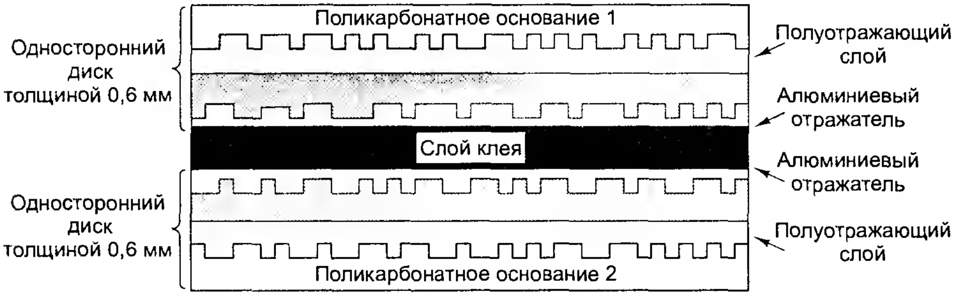


Рис. 5.19. Двусторонний двухуровневый DVD

Технология DVD была разработана консорциумом из 10 компаний-производителей бытовой электроники, семь из которых были японскими, в тесном сотрудничестве с голливудскими студиями (некоторыми из них владеют японские электронные компании, входящие в консорциум). Представителей компьютерной и телекоммуникационной промышленностей не пригласили на пикник, в результате чего акцент оказался смещен в сторону использования DVD для размещения на них фильмов в основном в коммерческих целях. Например, стандартные свойства DVD включают пропуск грязных сцен в режиме реального времени (что позволяет родителям превратить порнофильм в детскую сказку), шестиканальный звук и поддержку Pan-and-Scan. Последняя особенность позволяет DVD-проигрывателю динамически решать, как обрезать левый и правый края фильма (с соотношением сторон 3:2), чтобы кадр помещался на экран современного телевизора (с соотношением сторон 4:3).

Кроме того, представителям компьютерной промышленности, возможно, просто не пришло бы в голову намеренно добиваться несовместимости дисков, предназначенных для продажи на разных континентах. Эта «функция» была включена по требованию Голливуда, поскольку все голливудские фильмы сначала демонстрируются на широких экранах США, после чего их везут в Европу. К этому времени в США на полках магазинов уже появляются видеоверсии этих фильмов. Замысел состоял в том, чтобы гарантировать, что европейские видеомагазины не купят диски в США слишком рано, тем самым снизив доходы европейских кинотеатров. Если бы Голливуд занимался компьютерным бизнесом, у нас сейчас были бы 3,5-дюймовые дискеты в США и 9-см диски в Европе.

Форматирование дисков

Жесткий диск состоит из стопки сделанных из алюминия, стекла или других материалов пластин диаметра 5,25 дюйма или 3,5 дюйма (или даже меньших для ноутбуков). На каждую пластину нанесен тонкий магнитный слой оксида металла. После создания на диске нет никакой информации.

Прежде чем диском можно будет пользоваться, каждой пластине нужно программно придать **низкоуровневый формат**. Этот формат состоит из серий концентрических дорожек, содержащих определенное число секторов, с короткими промежутками между секторами. Формат сектора показан на рис. 5.20.

Заголовок	Данные	ECC

Рис. 5.20. Сектор диска

Заголовок начинается с определенной последовательности битов, обеспечивающей распознавание начала сектора аппаратурой. Он также содержит номера цилиндра и сектора и еще некоторую информацию. Размер порции данных определяется программой низкоуровневого форматирования. В большинстве дисков используются 512-байтовые секторы. Поле ECC (Error Correction Code — код корректировки ошибок, контрольная сумма) содержит избыточную информацию, позволяющую обнаруживать (и даже, возможно, исправлять) ошибки чтения. Размер и содержимое этого поля зависят от производителя диска и его желания увеличить емкость диска за счет снижения надежности или наоборот, а также способности контроллера поддерживать тот или иной алгоритм подсчета контрольной суммы. Не является редкостью, например, 16-байтовое поле ECC. Кроме того, все жесткие диски обладают некоторым количеством запасных секторов, используемых для замены секторов с производственными дефектами.

При низкоуровневом форматировании 0-й сектор располагается на каждой следующей дорожке со сдвигом относительно предыдущей дорожки. Это смещение, называемое **перекосом цилиндров**, служит увеличению производительности диска. Замысел состоит в том, чтобы диск мог читать несколько дорожек за одну операцию без потери времени на ожидание. Суть проблемы можно проиллюстрировать на рис. 5.14, *а*. Предположим, необходимо прочитать 18 секторов, начиная с сектора 0 внутренней дорожки. Чтение 16 секторов (полной дорожки) занимает один оборот диска, но для перехода на следующую дорожку требуется время. К тому моменту, когда считывающая головка будет установлена на нужное место, 17-й сектор (сектор 0 следующей дорожки) уже успеет провернуться под головкой, поэтому, чтобы его прочитать, придется ждать целый оборот диска. Эта проблема решается при помощи разметки секторов со смещением, как показано на рис. 5.21.

Величина перекоса цилиндров зависит от геометрии диска. Например, для диска, вращающегося со скоростью 10 000 об/мин, время одного оборота составляет 6 мс. Если на дорожке содержится 300 секторов, сектор проходит под головкой каждые 20 мкс. За время перемещения головки на соседнюю дорожку, равное 800 мкс, под головкой пройдут 40 секторов. Таким образом, перекос цилиндров должен составлять 40 секторов, а не три сектора, как показано на рис. 5.21. Следует также заметить, что переключение с одной головки на другую тоже занимает время, поэтому **перекос головок** присутствует, но он не так велик, как перекос цилиндров.

В результате низкоуровневого форматирования емкость диска немного уменьшается в связи с расходами на заголовки сектора, межсекторные промежутки и поля ECC, а также резервные секторы. Часто полезный объем жесткого диска составляет около 80 % от общего объема. Резервные секторы не входят в емкость отформатированного диска, поэтому у всех дисков одного типа при поставке абсолютно одина-

ковая емкость, независимо от числа дефектных секторов на дисках. (Если число дефектных секторов превышает количество запасных секторов, то такой диск не проходит контроль качества и не поставляется.)

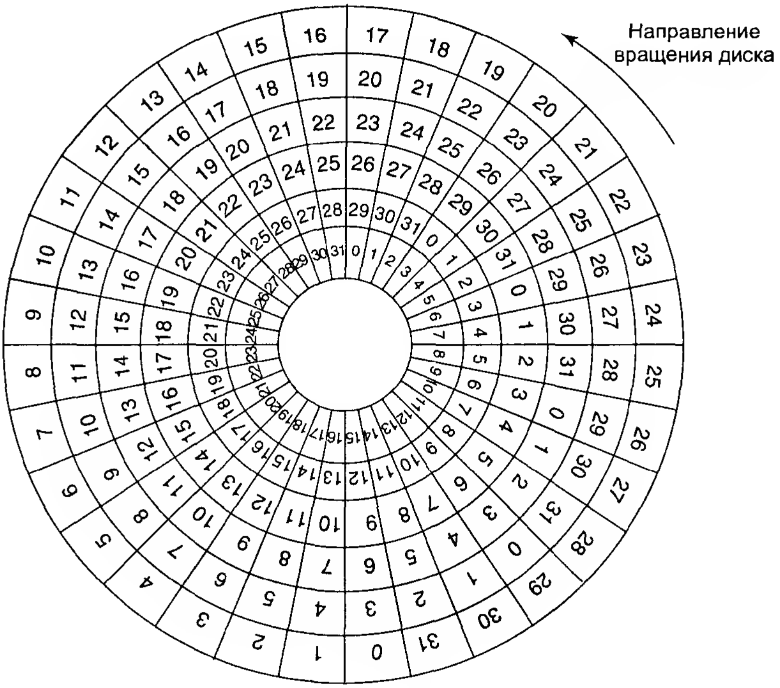


Рис. 5.21. Иллюстрация перекоса цилиндров

В вопросе емкости дисков существует определенная путаница, вызванная желанием производителей создать впечатление, что их диск больше, чем на самом деле. Поэтому иногда производителем может рекламироваться неформатированная емкость жесткого диска вместо форматированной. Например, возьмем диск с неформатированной емкостью 20×10^9 байт. Он может быть продан за 20-гигабайтный диск. Однако после форматирования для данных окажутся доступными только $2^{34} \approx 17{,}2 \times 10^9$ байт. Операционная система обычно считает гигабайты как 2^{30} байт, а не 10^9 байт. В результате она сообщит, что емкость диска составляет 16,0 Гбайт, а не 17,2 Гбайт.

Путаницы добавляют единицы измерения скорости передачи данных, где приставка «гига» снова означает 10^9, а не 2^{30}. Степени числа 2 используются для измерения объема памяти, дисков, размеров файлов, где кило, мега, гига и тера означают 2^{10}, 2^{20}, 2^{30} и 2^{40}. Еще запутаннее этот вопрос становится, когда скорость передачи данных измеряют то в килобитах в секунду (1000 бит/с), то в килобайтах в секунду (1024 байт/с).

Форматирование также влияет на производительность диска. Если у диска, вращающегося со скоростью 10 000 об/мин, на каждой дорожке по 300 секторов размером по 512 байт, то вся дорожка, то есть 153 600 байт, считывается за 6 мс,

что составляет скорость считывания данных 25 600 000 байт/с или 24,4 Мбайт/с. Быстрее считывать данные невозможно, независимо от используемого интерфейса, даже если это SCSI со скоростью передачи данных 80 Мбайт/с или 160 Мбайт/с.

Чтение с такой скоростью в течение длительного периода времени требует большого буфера в контроллере. Представьте контроллер с буфером размером в один сектор, которому дана команда прочитать два сектора подряд. Прочитав первый сектор с диска и проверив его контрольную сумму, контроллер должен передать данные в оперативную память. Пока первый сектор передается в оперативную память, второй сектор уже провернется под головкой, и контроллеру придется ждать почти полный оборот диска.

Эта проблема может быть решена, если при форматировании нумеровать секторы не подряд. На рис. 5.22, *а* показана обычная нумерация секторов. На рис. 5.22, *б* мы видим **однократное чередование** секторов, предоставляющее контроллеру возможность перевести дух и скопировать прочитанный сектор из буфера в оперативную память. Это позволяет снизить скорость считывания всего в два раза, а не, например, в 300 раз, если на одной дорожке помещается 300 секторов.

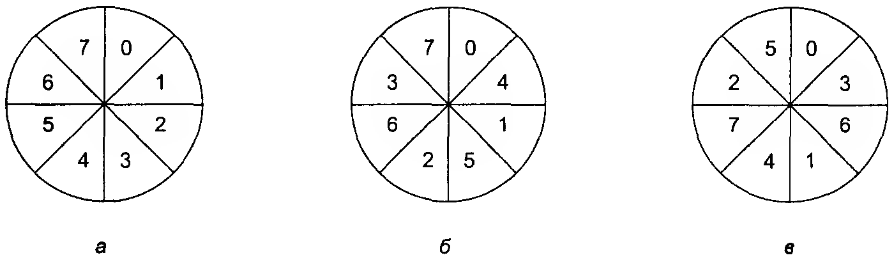


Рис. 5.22. Без чередования (*а*); однократное чередование (*б*); двукратное чередование (*в*)

Если процесс копирования очень медленный, может потребоваться **двукратное чередование**, показанное на рис. 5.22, *в*. Если у контроллера буфер только на один сектор, то не важно, выполняется ли копирование в оперативную память контроллером, центральным процессором или микросхемой DMA. На это все равно нужно время. Чтобы избежать необходимости в чередовании секторов, контроллер должен иметь достаточно большой буфер, чтобы прочитать в него сразу целую дорожку. У многих современных контроллеров такой буфер есть.

После выполнения низкоуровневого форматирования диск разбивается на разделы. Логически каждый раздел диска воспринимается операционной системой как отдельный диск. На компьютерах с процессором Pentium и большинстве других компьютеров в секторе 0 помещается **главная загрузочная запись**, содержащая часть загрузочной программы и таблицу разделов. Таблица разделов содержит номера начальных секторов и размеры каждого раздела. На Pentium-компьютерах в таблице разделов есть место для четырех разделов. Если все они предназначены для системы Windows, они будут называться устройствами C:, D:, E: и F:. Если на трех из них размещается система Windows, а на четвертом UNIX, тогда Windows будет называть свои разделы C:, D: и E: (четвертый раздел не будет виден системе Windows). Первое устройство чтения компакт-дисков будет назы-

ваться F:. Чтобы компьютер мог загружаться с жесткого диска, один из разделов должен быть помечен как активный.

Последним этапом подготовки диска к употреблению заключается в **высокоуровневом форматировании** каждого раздела (по отдельности). Эта операция помещает в раздел загрузочный блок, бит-карту или список свободных блоков устройства, корневой каталог и пустую файловую систему. Кроме того, в таблицу разделов помещается определенный код, указывающий, какая файловая система используется в данном разделе, поскольку многие операционные системы традиционно поддерживают несколько несовместимых файловых систем. Затем в одном из разделов может быть установлена операционная система.

При включении питания компьютера запускается базовая система ввода-вывода BIOS, которая считывает главную загрузочную запись с диска и передает в него управление. Загрузочная программа определяет, который из разделов диска является активным. Из этого раздела считывается и запускается загрузочный сектор. Загрузочный сектор содержит маленькую программу, которая находит в корневом каталоге определенный файл (либо операционную систему, либо загрузчик больших размеров). Этот файл загружается в память и запускается.

Алгоритмы планирования перемещения головок

В этом разделе мы рассмотрим некоторые вопросы, касающиеся в основном дисковых драйверов. Прежде всего определим, сколько времени занимает считывание или запись одного блока диска. Необходимое для этого время определяется тремя факторами:

1. Время поиска (время перемещения головки на нужный цилиндр).
2. Задержка вращения (время, требуемое для поворота нужного сектора под головку).
3. Время передачи данных.

Для большинства дисков первая составляющая (время поиска) существенно превосходит остальные две, поэтому значительного увеличения производительности системы можно добиться, снижая время поиска.

Если драйвер диска принимает и выполняет запросы по одному в порядке получения, то есть по принципу «первым пришел — первым обслужен» (FCFS, First Come, First Served), тогда мало что можно сделать для оптимизации времени поиска. Однако при сильной загрузке диска возможно применение другой стратегии. В этом случае высока вероятность поступления новых запросов от других процессов во время перемещения головки для обработки предыдущего запроса. Многие дисковые драйверы содержат таблицу, индексированную по номерам цилиндров, в которой в единый связный список собираются все поступившие и ждущие обработки обращения к цилиндрам.

С помощью подобной структуры данных можно создать более совершенный алгоритм планирования, чем простое обслуживание в порядке поступления запросов. Рассмотрим, например, диск с 40 цилиндрами. Поступает запрос на чтение блока с цилиндра 11. Во время перемещения головки на 11-й цилиндр поступают

новые обращения к цилиндрам 1, 36, 16, 34, 9 и 12. Новые запросы помещаются в таблицу, состоящую из отдельных списков для каждого цилиндра. Поступившие запросы помечены крестиками на рис. 5.23.

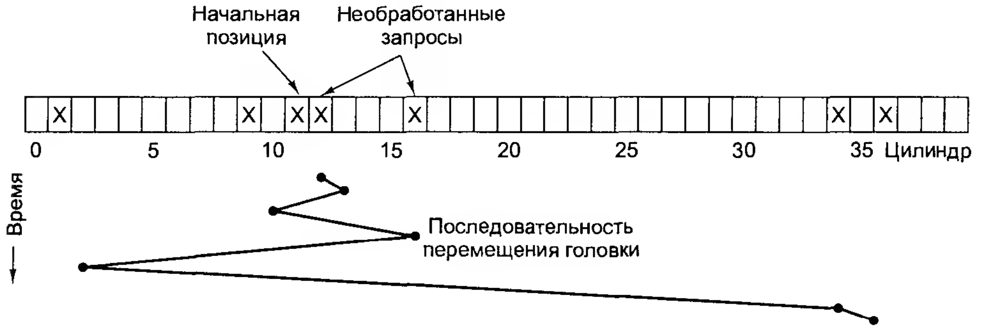


Рис. 5.23. Алгоритм планирования «ближайший цилиндр первым»

Выполнив первый запрос (обращение к цилиндру 11), драйвер диска должен выбрать следующий запрос. Если обслуживать запросы в порядке поступления, то он должен переместить головку на цилиндр 1, затем на цилиндр 36 и т. д. В результате выполнение этого алгоритма потребует перемещения блока головок на 10, 35, 20, 18, 25 и 3 цилиндра, что в сумме составит 111 цилиндров.

Время перемещения головки можно уменьшить, выбирая каждый раз ближайший цилиндр. При той же серии запросов, показанной на рис. 5.23, последовательность их обработки выглядит как ломаная кривая внизу рисунка. При такой последовательности выполнения запросов потребуются перемещения блока головок на 1, 3, 7, 15, 33 и 2 цилиндра, что в сумме составит 61 цилиндр. Применение данного алгоритма, называющегося **SSF** (Shortest Seek First — «ближайший запрос первым»), снижает суммарные перемещения головок почти вдвое.

К сожалению, у данного алгоритма есть свои недостатки. Предположим, во время обработки запросов, показанных на рис. 5.23, поступили новые запросы. Например, после обработки обращения к цилиндру 16 поступает новый запрос к цилиндру 8. Новый запрос будет иметь приоритет над обращением к цилиндру 1. Затем, если поступит запрос к цилиндру 13, то головка опять будет перемещаться к цилиндру 13 для обработки нового запроса, а запрос к цилиндру 1 останется необработанным. При сильной загрузке диска головка диска будет большую часть времени находиться где-то в районе средних цилиндров, а запросам к крайним цилиндрам диска придется ждать, пока нагрузка диска случайно не снизится и обращений к середине диска не станет меньше. В результате качество обслуживания запросов к цилиндрам, сильно удаленным от середины, может оказаться низким. То есть в конфликт вступают принцип минимизации времени отклика и принцип справедливости.

В высотных зданиях также приходится иметь дело с данной проблемой планирования обслуживания запросов лифта. Вызовы лифта постоянно поступают с разных этажей. Компьютер, управляющий лифтом, должен следить за последовательностью поступления запросов и обслуживать их либо в порядке подачи заявок, либо обслуживая первым ближайший запрос.

Однако в большинстве лифтов применяются различные алгоритмы, пытающиеся примирить конфликтующие цели эффективности и справедливости. Обычно лифт продолжает двигаться в одном направлении до тех пор, пока на этом направлении более не остается запросов. Затем лифт меняет направление движения. Этот алгоритм, называющийся **элеваторным алгоритмом**, требует от программного обеспечения следить всего за одним битом, хранящим информацию о текущем направлении движения, *ВВЕРХ* или *ВНИЗ*. Выполнив очередной запрос, диск или лифт проверяет значение бита. Если в нем содержится значение *ВВЕРХ*, кабина лифта или блок головок перемещается вверх к следующему запросу. Если в этом направлении запросов больше нет, состояние бита инвертируется.

Рисунок 5.24 иллюстрирует работу элеваторного алгоритма с теми же семью запросами, что были использованы в качестве примера на рис. 5.23. Предполагается, что изначально бит направления движения указывал *ВВЕРХ*. При этом цилиндры обслуживаются в порядке 12, 16, 34, 36, 9 и 1, что соответствует перемещению блока головок на 1, 4, 18, 2, 27 и 8 цилиндров и в сумме составляет 60 цилиндров. В данном случае элеваторный алгоритм оказался даже слегка лучше, чем алгоритм SSF, однако на практике он обычно несколько хуже. Достоинство элеваторного алгоритма состоит в том, что при любом заданном наборе запросов верхняя граница необходимых перемещений блока головок для выполнения всех запросов фиксирована и ограничена двойным числом цилиндров.

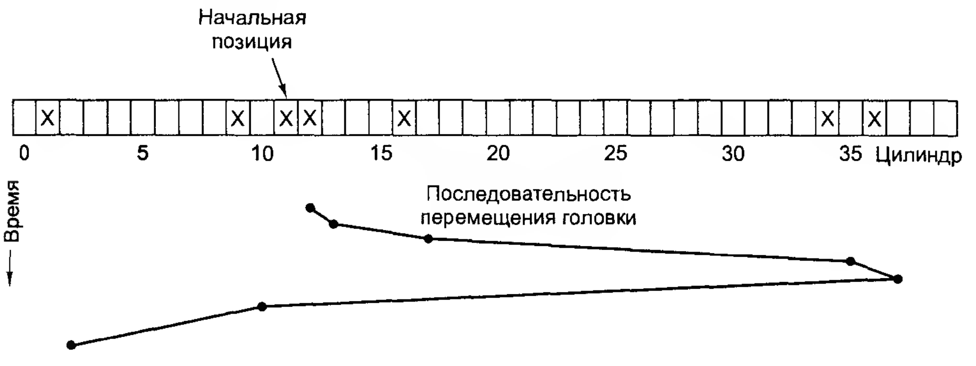


Рис. 5.24. Элеваторный алгоритм планирования обращений к диску

Незначительная модификация этого алгоритма с меньшим разбросом времени отклика состоит в том, чтобы сканировать цилиндры всегда в одном направлении [328]. После обслуживания обращения к цилиндру с самым высоким номером блок головок перемещается к самому нижнему запрашиваемому цилиндру, после чего продолжает движение вверх. Таким образом, самый нижний цилиндр как бы считается расположенным выше самого верхнего.

Некоторые контроллеры дисков предоставляют программному обеспечению способ узнавать номер текущего сектора под головкой. Такие контроллеры позволяют использовать дополнительный метод оптимизации. Если есть два или более запросов к одному и тому же цилиндру, драйвер может выбрать из них тот, сектор которого пройдет под головкой первым. Обратите внимание, что при наличии нескольких головок у диска последовательные запросы могут обращаться к различ-

ным дорожкам на одном цилиндре. Контроллер может практически мгновенно переключаться с головки на головку, так как такое переключение не требует ни перемещения блока головок, ни ожидания поворота диска.

Если у диска время перемещения головок значительно ниже задержки, связанной со скоростью вращения диска, тогда следует применять другую стратегию оптимизации. Запросы, ожидающие обработки, должны сортироваться по номерам секторов, и как только следующий сектор приблизится к головке, блок головок должен быстро переместиться на нужную дорожку, чтобы считать или записать его.

У современных жестких дисков производительность настолько сильно определяется задержками перемещения головок и вращения диска, что читать по одному или по два сектора крайне неэффективно. По этой причине многие современные контроллеры дисков читают и сохраняют в кэше по нескольку секторов сразу, даже если запрашивается всего один сектор. Обычно при этом считывается запрашиваемый сектор и все остальные секторы этой дорожки, при условии, что для них достаточно места в кэше контроллера. Например, у диска, описанного в табл. 5.3, размер кэша обычно составляет 2 Мбайт или 4 Мбайт. Режим использования кэша динамически определяется контроллером. В простейшем режиме кэш разделяется на две секции, одна для запросов чтения, другая для запросов записи. Если последующий запрос на чтение сектора может быть удовлетворен прямо из кэша контроллера, запрашиваемые данные могут быть выданы немедленно.

Следует отметить, что кэш контроллера диска абсолютно никак не связан с кэшем операционной системы. Кэш контроллера обычно содержит блоки, на которые запрос еще не поступал, но которые было удобно прочитать, так как они случайно оказались под головкой при чтении других блоков. Напротив, любой кэш операционной системы состоит из блоков, на чтение которых были явно направлены запросы и которые, по мнению операционной системы, могут понадобиться снова в ближайшем будущем (например, блок диска, содержащий каталог).

При одновременном обслуживании контроллером нескольких устройств таблица запросов, ждущих обработки, должна поддерживаться отдельно для каждого устройства. Когда любое из устройств завершает выполнение текущего запроса, ему нужно дать команду на перемещение блока головок на цилиндр для выполнения следующего запроса (при условии, что контроллер позволяет работать в режиме перекрывающегося поиска). По завершении текущей операции переноса данных может быть выполнена проверка правильного позиционирования блоков головок всех устройств. Если хотя бы один блок установлен, может быть начат следующий перенос данных. В противном случае драйвер должен выдать новую команду поиска цилиндра устройству, только что завершившему операцию переноса данных, после чего перейти в состояние ожидания прерывания от диска, установившего блок головок на нужный цилиндр и готового к перемещению данных.

Важно понимать, что во всех вышеизложенных алгоритмах планирования работы дисков молчаливо подразумевается, что физическая геометрия дисков совпадает с виртуальной. В противном случае планирование обращений к диску не имеет никакого смысла, так как операционная система не сможет определить, какой цилиндр расположен ближе к цилиндру 39, 40-й или 200-й. С другой стороны, если контроллер диска способен принимать несколько внешних запросов одновременно, он может выполнять все планирование внутренне. В этом случае алгоритмы сохраняют силу, но выполняются на боле низком уровне, в контроллере.

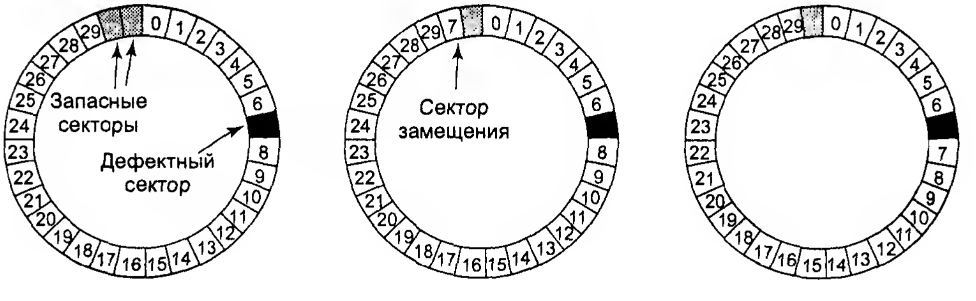
Обработка ошибок

Производители дисков постоянно раздвигают технологические пределы, увеличивая линейную плотность битов. Окружность средней дорожки на 5,25-дюймовом жестком диске составляет около 300 мм. Если эта дорожка содержит 300 секторов по 512 байт, то, с учетом потерь на межсекторные промежутки, заголовки секторов и поля ECC, линейная плотность записи может составлять около 5000 бит/мм. Такая высокая плотность записи требует крайне высокой степени гладкости поверхности диска и тонкости магнитного покрытия. К сожалению, создать диск с такими высокими характеристиками без дефектов невозможно. К тому моменту, когда производственные технологии совершенствуются настолько, что позволяют создавать диски с такой плотностью без дефектов, проектировщики дисков увеличивают плотность записи для увеличения емкости дисков. Таким образом, диски, по-видимому, всегда будут содержать определенное количество дефектов.

Производственные дефекты проявляются в виде дефектных секторов, то есть секторов, которые не читаются правильно после записи. Если дефект сектора невелик, например всего несколько бит, такой сектор можно использовать, позволив полю ECC исправлять ошибки при каждом чтении сектора. При более серьезных дефектах ошибка уже не может быть скорректирована.

Дефектные блоки могут обрабатываться контроллером или операционной системой. При первом подходе каждый диск проверяется на фабрике, а список дефектных секторов записывается на диск. Вместо каждого дефектного блока используется запасной.

Подобная замена может выполняться двумя способами. На рис. 5.25, *а* показана дорожка диска из 30 блоков и 2 запасных. Сектор 7 поврежден. Контроллер может пометить один из запасных секторов как сектор 7, как показано на рис. 5.25, *б*. Другой способ состоит в сдвиге всех секторов на один сектор (рис. 5.25, *в*). В обоих случаях контроллер должен знать номера каждого сектора. Он должен хранить эту информацию либо в своих внутренних таблицах (по одной на дорожку), либо перезаписывая заголовки секторов. При перезаписи заголовков метод на рис. 5.25, *в* требует большей работы (так как для этого нужно перезаписать 23 заголовка), но в конечном итоге он дает лучшую производительность, так как при этом вся дорожка все так же может быть прочитана за один оборот.



а *б* *в*

Рис. 5.25. Дорожка диска с дефектным сектором (а); замена дефектного сектора запасным (б); сдвиг всех секторов (в)

Ошибки могут возникать при выполнении обычных операций после установки диска. Первая линия обороны при получении ошибки, которую не может исправить ECC, состоит в простом повторении попытки чтения. Некоторые ошибки чтения вызываются попаданием пылинки на головку и исчезают при повторной попытке. Если контроллер замечает повторяющиеся ошибки на определенном секторе, он может переключиться с него на запасной сектор раньше, чем сектор с ошибками выйдет из строя окончательно. При этом данные не будут потеряны, а операционная система и пользователь даже не заметят наличия проблемы. Для этого обычно применяется метод, показанный на рис. 5.25, *б*, так как остальные секторы этой дорожки могут уже содержать данные. Использование метода, изображенного на рис. 5.25, *в*, потребует перезаписи не только всех заголовков, но также и копирования всех данных.

Как говорилось ранее, есть два основных подхода к исправлению ошибок: обработка их контроллером или операционной системой. Если контроллер не может прозрачно преобразовывать секторы, тогда этим должна программно заниматься операционная система. Это означает, что сначала она должна получить список всех дефектных секторов, либо прочитав этот список с диска, либо самостоятельно проверив весь диск. Узнав, какие секторы являются дефектными, она может создать таблицы соответствий. Если преобразованием занимается операционная система, она должна гарантировать, что дефектные секторы не встречаются в файлах, а также в списке или битовом массиве свободных секторов. Один из способов добиться этого состоит в создании секретного файла, включающего в себя все дефектные секторы. Если этот файл не является частью файловой системы, пользователи не смогут его случайно прочитать или, что еще хуже, удалить.

Однако есть еще одна проблема, связанная с созданием резервных копий диска. Если архивация диска выполняется по файлам, то важно, чтобы программа архивации не пыталась копировать файл дефектных блоков. Для этого операционная система должна спрятать этот файл настолько хорошо, чтобы даже архивирующая программа не смогла его найти. Если же диск архивируется посекторно, а не пофайлово, то будет трудно, если вообще возможно, избежать ошибок чтения во время архивации. Единственная надежда на то, что архивирующей программе достанет здравого смысла прекратить попытки чтения дефектного сектора после 10 попыток и перейти к чтению следующего сектора.

Дефектные секторы не являются единственным источником ошибок. Также возникают ошибки поиска цилиндра, вызванные механическими проблемами блока головок. Контроллер следит за положением блока головок. При установке головки на заданный цилиндр он выдает серию импульсов двигателю блока головок, по одному импульсу на цилиндр. Когда блок головок устанавливается в требуемое положение, контроллер считывает истинное значение цилиндра из заголовка первого попавшегося сектора. Если блок головок оказывается не на той дорожке, на которой нужно, возникает ошибка поиска.

Большинство контроллеров жестких дисков автоматически исправляют ошибки поиска, но большинство контроллеров гибких дисков (включая установленные на компьютерах с процессором Pentium) просто выставляют бит ошибки и оставляют все остальное драйверу. Драйвер обрабатывает ошибку, издавая команду `recalibrate`. При этом блок головок отодвигается на самую внешнюю дорожку дис-

ка до упора. Это положение принимается контроллером за нулевую дорожку, таким образом, контроллер снова начинает понимать, где находится блок головок. Обычно это решает проблему. Если же нет, то либо нужно сменить диск, либо починить дисковод.

Как мы видели, контроллер в действительности представляет собой небольшой специализированный компьютер, полный программного обеспечения, переменных, буферов и, по всей видимости, ошибок. Иногда необычное стечение обстоятельств, например приход прерывания с одного устройства, в то время как другое устройство выполняет команду recalibrate, может разбудить спящую до тех пор ошибку, в результате которой контроллер может войти в бесконечный цикл или забыть, что он делал. Разработчики контроллеров обычно готовятся к худшему и предоставляют на всякий случай специальный контакт на микросхеме, обращение к которому вызывает сброс контроллера. Если никакие другие меры не помогают, драйвер может установить этот бит, приводящий к подаче сигнала на такой контакт, и сбросить контроллер. Если и это не помогает, то все, что драйверу остается сделать, — это вывести сообщение и сдаться.

При повторной калибровке дисковод издает скрипучий звук, но в остальном эта процедура обычно не очень сильно беспокоит. Однако имеются ситуации, в которых повторная калибровка представляет собой серьезную проблему: системы с ограничениями реального времени. При воспроизведении видео с жесткого диска или при перезаписи файлов с винчестера на CD-R крайне важно, чтобы биты поступали с жесткого диска с постоянной скоростью. В этих случаях повторная калибровка может нарушить непрерывность потока битов и поэтому является неприемлемой. Для подобных приложений существуют специальные накопители, называющиеся **AV-дисками** (Audio Visual disks — аудио видео диски), которые никогда не выполняют операцию повторной калибровки.

Стабильное запоминающее устройство

Как мы видели, диски иногда ошибаются. Хорошие секторы могут внезапно стать дефектными. Целый диск внезапно может выйти из строя. Чередующиеся наборы данных RAID могут защитить от выхода из строя нескольких секторов или целого диска. Однако они не могут защитить от сбоев во время записи, при которых записываются неверные данные или, что еще хуже, данные записываются не туда, куда надо, уничтожая другие важные данные.

Для некоторых данных крайне важно, чтобы данные никогда не были потеряны или испорчены, даже по причине ошибок диска или центрального процессора. В идеале диск должен просто всегда работать без ошибок. К сожалению, это недостижимо. Однако можно создать дисковую подсистему, обладающую следующими свойствами: получив команду записи, такое устройство либо корректно записывает данные, либо не делает ничего, не изменяя хранящиеся на нем данные. Подобная система называется **стабильным запоминающим устройством** и реализуется программно [195]. Ниже будет описана разновидность оригинальной идеи.

Прежде чем создать алгоритм, важно иметь ясную модель всех возможных ошибок. Модель предполагает, что при записи блока (одного или нескольких секторов) эта операция записи либо корректна, либо некорректна, и эта ошибка может

быть обнаружена с помощью последующего чтения и изучения полей ЕСС. Гарантированное обнаружение ошибки невозможно в принципе, так как 512-байтовый сектор, способный хранить 2^{4096} различных комбинаций данных, защищается 16-байтовым полем ЕСС, способным принимать всего 2^{128} значений[1], из которых допустимыми являются далеко не все. Таким образом, существуют миллиарды миллиардов комбинаций (из которых всего одна верная), соответствующих одному и тому же значению поля ЕСС. Вероятность случайного совпадения значения 16-байтового поля ЕСС составляет около 2^{-128}, то есть настолько мало, что мы можем называть это число нулем, хотя в действительности это не так.

В модели также учитывается, что правильно записанный сектор может внезапно стать дефектным и перестать читаться. Однако предполагается, что такие события происходят столь редко, что вероятность выхода из строя одинаковых секторов на основном и резервном дисках за небольшой интервал времени (например, один день) можно смело считать равной нулю.

Модель также допускает выход из строя центрального процессора. В этом случае он просто останавливается. Любая операция записи, совершающаяся в этот момент, также останавливается, это приводит к неверным данным в одном секторе и неверному значению поля ЕСС, что может быть обнаружено позднее. При всех вышеперечисленных условиях возможно создать 100-процентно надежное стабильное запоминающее устройство, то есть либо записывающее данные корректно, либо оставляющее все хранящиеся данные так, как есть. Конечно, такое устройство не может противостоять землетрясениям или защитить данные от падения компьютера со стометровой высоты.

Стабильное запоминающее устройство использует пару идентичных дисков, в которых соответствующие блоки работают совместно, образуя блоки, защищенные от ошибок. При отсутствии ошибок соответствующие блоки на обоих дисках идентичны. Для получения одинакового результата может быть прочитан любой из дисков. Для достижения этой цели определены три следующие операции.

1. **Стабильная операция записи.** Стабильная операция записи выглядит следующим образом. Сначала на диск 1 записывается блок данных, который затем считывается для проверки корректности выполнения этой операции. Если обнаруживается ошибка, операции записи и чтения повторяются в цикле до тех пор, пока при чтении блока не будет ошибки или пока этот цикл не будет выполнен определенное число раз. Если после выполнения заданного числа циклов операций записи и чтения успешного результата добиться не удается, блок диска помечается как дефектный и вместо него используется резервный блок. После этого вся процедура повторяется до тех пор, пока блок не будет записан, независимо от того, сколько резервных блоков придется задействовать. Когда наконец удается записать блок на диск 1 с использованием той же процедуры, записывается и проверяется соответствующий ему блок диска 2. Если повезет и центральный процессор не выйдет из строя до завершения всей операции, блок будет корректно записан и проверен на обоих дисках.

[1] У автора здесь дважды упоминается 2^{144} и 2^{-144}, что не может быть верным, так как 16 байт — это 128 бит, а не 144. — *Примеч. перев.*

2. **Стабильная операция чтения.** При этой операции сначала считывается блок с диска 1. Если проверка значения поля ECC не дает верного результата, считывание блока и проверка контрольной суммы повторяются в цикле определенное число раз. Если все попытки чтения оказываются неуспешными, соответствующий блок читается с диска 2. Поскольку операция стабильной записи создает две проверенные копии блока, а также предполагается, что вероятность внезапного выхода из строя сразу двух соответствующих блоков за короткий интервал времени пренебрежимо мала, операция стабильного чтения всегда завершается успешно.
3. **Восстановление от сбоев.** После сбоя программа восстановления сканирует оба диска и сравнивает соответствующие блоки. Если оба блока успешно читаются и совпадают, то никаких действий не выполняется. Если у одного из блоков при чтении обнаруживается ошибка контрольной суммы (ECC), то на место дефектного блока записывается правильный блок. Если оба блока читаются без ошибки, но не совпадают, тогда блок с диска 1 пишется поверх блока на диске 2.

При отсутствии сбоев центрального процессора эта схема всегда работает, так как при операции стабильной записи блок всегда записывается дважды, а также предполагается, что сразу два соответствующих блока никогда не выходят из строя одновременно. Как может повлиять сбой центрального процессора на операцию стабильной записи? Результат зависит от того момента, когда произойдет сбой. Возможны пять вариантов, показанные на рис. 5.26.

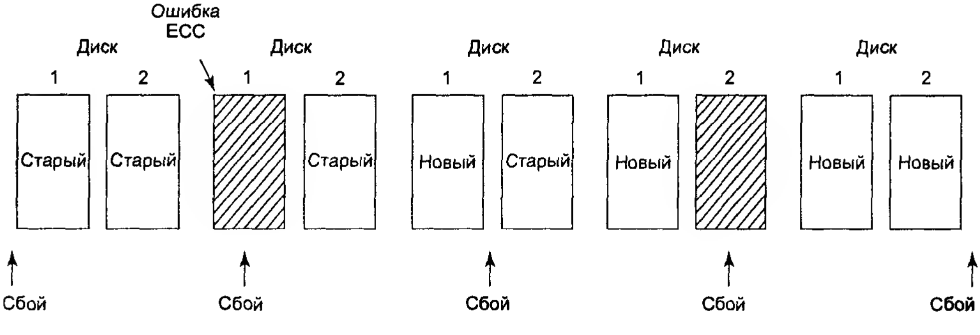


Рис. 5.26. Анализ влияния сбоя процессора на стабильность записи

На рис. 5.26, *а* сбой центрального процессора происходит прежде, чем записывается какая-либо копия блока. При восстановлении ничего не меняется и сохраняется старое значение, что является допустимым.

На рис. 5.26, *б* сбой центрального процессора происходит во время записи блока на диск *1*, в результате чего разрушается содержимое этого блока. Программа восстановления обнаруживает эту ошибку и восстанавливает блок на диске *1* с диска *2*. Таким образом, результат сбоя устраняется и восстанавливается старое значение.

На рис. 5.26, *в* сбой центрального процессора происходит после записи на диск *1*, но до записи на диск *2*. Программа восстановления копирует блок с диска *1* на диск *2*. Операция записи считается успешной.

Ситуация на рис. 5.26, *г* похожа на ситуацию на рис. 5.26, *б*. При восстановлении дефектный блок заменяется правильным блоком. Окончательным значением обоих блоков будет новое значение.

Наконец, на рис. 5.26, *д*, как и в ситуации на рис. 5.26, *а*, программа восстановления видит, что оба блока успешно читаются и совпадают. Поэтому тут также ничего не изменяется.

К данной схеме применимы различные методы оптимизации и усовершенствования. Прежде всего, сравнение всех блоков, конечно, выполнимо, но все же занимает слишком много времени. Значительно ускорить этот процесс можно, записывая номера записываемых блоков перед операцией стабильной записи. В результате после сбой нужно будет проверить состояние только этих блоков. Это может быть реализовано несколькими различными способами. На некоторых компьютерах имеется небольшое количество **энергонезависимого ОЗУ**, представляющего собой специальную CMOS-память (Complementary Metal-Oxide Semiconductor — комплементарный металло-оксидный полупроводник), питающуюся от отдельной литиевой батареи. Такие батареи служат помногу лет, возможно, даже в течение всей жизни компьютера. В отличие от оперативной памяти, содержимое которой теряется после сбоя, содержимое энергонезависимого ОЗУ сохраняется. В энергонезависимом ОЗУ обычно хранится время и дата (изменяющиеся специальной микросхемой). В результате в компьютерах часы идут даже тогда, когда питание компьютера выключено.

Предположим, что несколько байтов энергонезависимого ОЗУ свободны и доступны операционной системе. Операция стабильной записи может поместить в него номер блока, который будет записываться. После успешного завершения операции стабильной записи на место номера блока в энергонезависимое ОЗУ записывается число, не соответствующее никакому номеру блока, например –1. Таким образом, после сбоя программа восстановления может проверить состояние энергонезависимого ОЗУ, чтобы определить, не находилась ли во время сбоя операция стабильной записи в процессе выполнения, и если да, то какой блок записывался. После этого остается проверить всего два блока на правильность и совпадение.

Если энергонезависимое ОЗУ недоступно, его можно имитировать следующим образом. В начале выполнения операции стабильной записи в некий фиксированный блок диска *1* записывается номер записываемого блока. Затем этот блок считывается для проверки. Когда этот блок успешно записан, записывается и проверяется также соответствующий блок диска *2*. Опять же в случае сбоя при записи номера блока всегда можно определить, находилась ли в момент сбоя операция стабильной записи в состоянии выполнения. Конечно, такой метод имитирования энергонезависимого ОЗУ требует дополнительного выполнения восьми дисковых операций при стабильной записи каждого блока, поэтому пользоваться таким методом нужно лишь в крайнем случае.

Следует обратить внимание еще на один момент. Мы предположили, что два соответствующих блока на разных дисках не смогут стать дефектными за один день. Если же пройдет много дней, то такое вполне может произойти. Поэтому раз в сутки должно выполняться полное сканирование обоих дисков с исправлением

всех ошибок. Таким образом, например, каждое утро оба диска будут гарантированно идентичными. Тогда если даже оба блока пары выйдут из строя, но с интервалом в несколько дней, все ошибки будут исправлены корректно.

Таймеры

Таймеры (также называемые **часами**) очень важны для работы любой многозадачной системы по ряду причин. Среди многих других задач, они следят за временем суток и не позволяют одному процессу надолго занять центральный процессор. Программное обеспечение таймера может принимать форму драйвера устройства, несмотря на то, что таймер не является ни блочным устройством вроде диска, ни символьным устройством типа мыши. Наше изучение таймеров будет проходить по тому же сценарию, что и предыдущие разделы: сначала мы рассмотрим аппаратную часть таймеров, а затем познакомимся с программным обеспечением.

Аппаратная часть таймеров

В компьютерах широко применяются два типа таймеров. Обе схемы сильно отличаются от наручных и настольных часов. Наиболее простые компьютерные часы привязываются по частоте к линии питания переменного напряжения 110 или 220 В и вызывают прерывания при каждом цикле напряжения с частотой 50 или 60 Гц. Такие часы очень широко применялись ранее, но сейчас являются редкостью.

Другой тип часов состоит из трех компонентов: кварцевого генератора, счетчика и регистра хранения, как показано на рис. 5.27. Если взять кусок кристалла кварца правильного размера и установить его в оправу под давлением, то можно заставить его колебаться и выдавать электрический сигнал с частотой в несколько сот мегагерц. Частота зависит от конкретного кристалла, но каждый кристалл выдерживает эту частоту с достаточно высокой точностью. С помощью электроники эту частоту можно поднять до 1 ГГц или даже до еще более высокой частоты. По крайней мере, одна такая схема обязательно присутствует в каждом компьютере, обеспечивая сигнал синхронизации для различных цепей компьютера. Этот сигнал подается на вход декрементного счетчика. Когда содержимое счетчика достигает нуля, он вызывает прерывание центрального процессора.

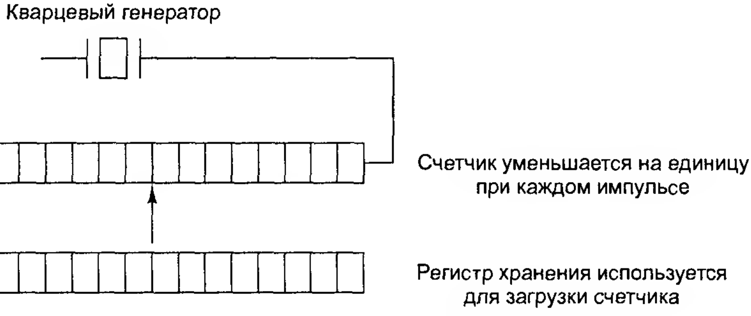


Рис. 5.27. Программируемый таймер

У программируемого таймера обычно есть несколько режимов работы. В режиме **одновибратора** при запуске таймера содержимое регистра хранения копируется в счетчик. Затем содержимое счетчика уменьшается на единицу при каждом импульсе от кристалла. Когда счетчик достигает нуля, он вызывает прерывание и останавливается до тех пор, пока он не будет снова явно запущен программным обеспечением. В режиме **генератора прямоугольных импульсов** при достижении счетчиком нуля инициируется прерывание, а содержимое регистра хранения автоматически копируется в счетчик, и весь процесс повторяется снова бесконечно.

Преимущество программируемого таймера состоит в том, что частота прерываний от него может управляться программно. Если используется кристалл с частотой колебаний 500 МГц, то счетчик получает импульс каждые 2 нс. При использовании 32-разрядного регистра можно запрограммировать возникновение прерываний через равные интервалы времени от 2 нс до 8,6 с, называемые **тиками**. Микросхемы программируемых таймеров обычно содержат два или три независимо программируемых счетчика и помимо этого обладают целым рядом других функций (например, могут увеличивать, а не уменьшать значение счетчика, не инициировать прерываний и т. д.).

Чтобы показания таймера не терялись, пока питание компьютера выключено, часы большинства компьютеров питаются от аккумулятора. Показания часов считываются при загрузке операционной системы. Если таких часов у компьютера нет, операционная система может запросить дату и время при запуске. Кроме того, система может узнать эти сведения по сети от удаленного хоста. В любом случае эти время и дата транслируются в количество интервалов таймера с какого-либо момента, например полуночи 1 января 1970 года по **всеобщему скоординированному времени** (UTC, Universal Coordinated Time), как это делает, например, система UNIX. До 1928 года время UTC называлось средним временем по Гринвичу (GMT, Greenwich Mean Time). В системе Windows время отсчитывается от 1 января 1980 года. При каждом прерывании от таймера счетчик времени увеличивается на единицу. В операционной системе обычно присутствуют программы, позволяющие скорректировать показания системных часов.

Программное обеспечение таймеров

Все, что делает таймер, аппаратно — он инициирует прерывания через определенные интервалы времени. Все остальное, связанное со временем, должно выполняться программно драйвером часов. Обязанности драйвера часов варьируются в зависимости от операционной системы, но обычными являются следующие функции:

1. Следят за временем суток.
2. Не позволяют процессам работать дольше, чем им разрешено.
3. Ведут учет использования центрального процессора.
4. Обрабатывают системный вызов `alarm`, инициированный процессом пользователя.
5. Поддерживают следящие таймеры для операционной системы.
6. Ведут наблюдение, анализ и сбор статистики.

Первая функция часов, поддерживающая время суток (также называемое **истинным временем**), не сложна. Она просто требует увеличения счетчика на единицу при каждом импульсе сигнала времени часов (рис. 5.28, *а*). Нужно только следить за количеством битов в счетчике времени суток. При частоте импульсов сигнала времени 60 Гц 32-разрядный счетчик переполнится уже за два года. Очевидно, система не может хранить значение истинного времени в тиках с 1 января 1970 года в 32 бит.

Для данной проблемы возможны три решения. Во-первых, можно использовать 64-разрядный счетчик, хотя это потребует больших затрат, так как увеличивать значение счетчика придется помногу раз в секунду (рис. 5.28, *б*). Второй способ состоит в хранении времени суток не в тиках (количестве импульсов сигнала времени), а в секундах, переводя импульсы сигнала времени в секунды при помощи дополнительного счетчика (рис. 5.28, *б*). Поскольку 2^{32} с — это больше, чем 136 лет, такой метод будет работать вплоть до 22-го века.

Третий метод состоит в том, чтобы учитывать импульсы сигнала времени, но относительно того момента, в который была загружена машина, а не от фиксированного внешнего момента. При этом система во время загрузки узнает текущее время, которое сохраняет в памяти в любом удобном виде. Позднее, при запросе времени, система складывает хранящееся время загрузки со значением счетчика, чтобы получить текущее время (рис. 5.28, *в*).

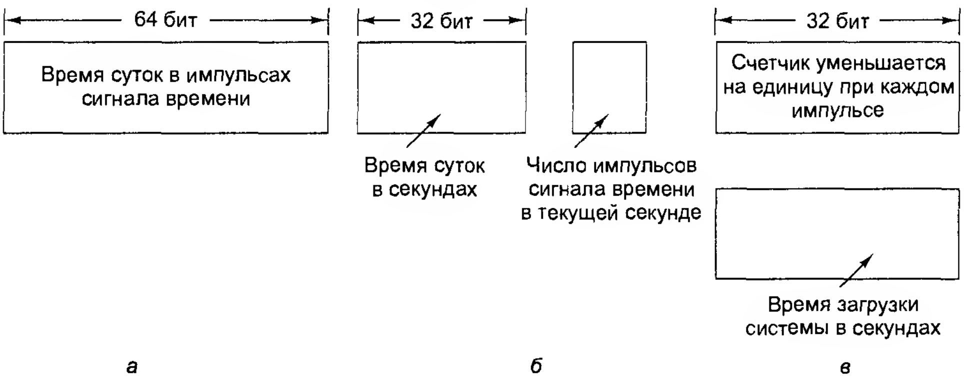


Рис. 5.28. Три способа реализации времени суток

Вторая функция часов состоит в недопущении слишком долгой работы процесса. При запуске процесса планировщик инициализирует счетчик, записывая в него выделенное этому процессу количество импульсов сигнала времени. При каждом прерывании от таймера драйвер таймера уменьшает значение счетчика на 1. Когда значение счетчика достигает нуля, драйвер таймера вызывает планировщик, чтобы тот запустил другой процесс.

Третья функция часов состоит в учете использования центрального процессора. Наиболее точно это может быть сделано, если при каждом запуске нового процесса запускать второй таймер, независимый от основных системных часов. Когда процесс останавливается, значение таймера считывается, чтобы определить, сколько времени работал процесс. Чтобы все было правильно, значения второго таймера должны сохраняться на время прерываний.

Не столь точный, но более простой метод учета состоит в создании указателя текущего процесса в таблице процессов в виде глобальной переменной. При каждом импульсе сигнала времени поле текущего процесса в таблице увеличивается на 1. Таким образом, каждый импульс сигнала времени «заботится» о текущем процессе. Недостаток этого метода состоит в том, что в случае частых прерываний во время работы процесса ему все равно будет засчитана работа в течение полного импульса сигнала времени. Точный учет использования времени центрального процессора во время прерываний является слишком сложным делом, отнимающим, в свою очередь, много процессорного времени.

Во многих системах процесс может попросить операционную систему выдать ему сигнал предупреждения после определенного интервала времени. Предупреждение может быть сигналом, прерыванием, сообщением и т. п. Такие предупреждения нужны, например, для работы в сети, при которой пакет, не получивший подтверждения в течение определенного интервала времени, должен быть передан повторно. Другим приложением может быть обучающая программа, ожидающая ответа на вопрос в течение установленного интервала времени.

Если драйвер часов управляет достаточным количеством таймеров, он может установить таймер для каждого запроса. Если физических таймеров недостаточно, они легко могут быть смоделированы программно. Один из способов реализации большого числа виртуальных таймеров состоит в создании таблицы, хранящей все времена сигналов для обрабатываемых таймеров, а также переменная, в которой хранится время срабатывания ближайшего таймера. При каждом обновлении времени суток драйвер проверяет, не пора ли подавать сигнал от ближайшего таймера. При этом ищется следующий по времени таймер.

Если ожидается много сигналов, то более эффективным считается реализовать их в виде сортированного связного списка, как показано на рис. 5.29. Каждый элемент списка содержит число импульсов сигнала времени относительно предыдущего таймера. В данном примере сигналы должны быть поданы в моменты времени 4203, 4207, 4213, 4215 и 4216.

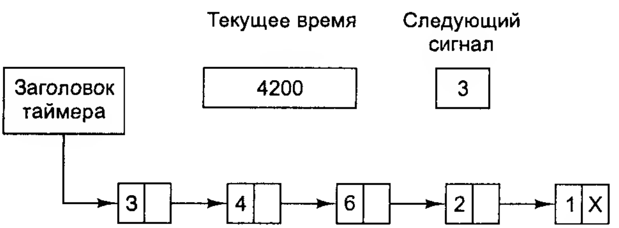


Рис. 5.29. Моделирование нескольких виртуальных таймеров

На рис. 5.29 следующее прерывание произойдет через 3 тика. На каждом тике значение переменной *Next signal*, хранящей число тиков, оставшееся до подачи следующего сигнала, уменьшается на 1. Когда значение этой переменной достигает нуля, подается сигнал в соответствии с первым элементом списка, который затем удаляется из списка. После этого переменной *Next signal* присваивается значение следующего элемента списка, то есть 4 в нашем примере.

Обратите внимание, что за время прерывания от таймера драйвер часов должен выполнить несколько действий: увеличить показания часов истинного времени, уменьшить значение кванта времени, выделенного текущему процессу, и сравнить его с нулем, выполнить операцию учета использования центрального процессора и уменьшить счетчик таймера тревоги. Однако все эти операции должны быть тщательно оптимизированы по времени исполнения, так как они будут повторяться много раз в секунду.

Операционной системе также требуются таймеры. Они называются **сторожевыми таймерами**. Например, гибкие диски не вращаются, пока ими не пользуются, чтобы избежать слишком быстрого изнашивания носителей и головок дисковода. Когда требуются данные с гибкого диска, следует запустить двигатель. Только если гибкий диск вращается с полной скоростью, может начаться операция ввода-вывода. Когда процесс пытается читать данные с находящегося в состоянии покоя гибкого диска, драйвер НГМД запускает двигатель и устанавливает сторожевой таймер, чтобы тот инициировал прерывание спустя время, достаточное для разгона диска. Сторожевой таймер необходим, так как накопители на гибких дисках не умеют формировать прерывания, сообщающие о том, что гибкий диск достаточно разогнался.

Механизм обработки сторожевых таймеров, используемый драйвером часов, тот же, что применяется для сигналов пользователя. Единственное отличие состоит в том, что когда таймер срабатывает, вместо подачи сигнала драйвер часов вызывает процедуру, предоставляемую обратившимся к нему процессом. Эта процедура является частью процесса. Она может сделать все, что нужно, даже вызвать прерывание, хотя внутри ядра прерывания часто бывают неудобны, а сигналов не существует. Вот почему предоставляется механизм сторожевых таймеров. Следует заметить, что этот механизм работает, только если драйвер таймера и вызываемая им процедура находятся в одном адресном пространстве.

Последняя функция таймеров в нашем списке — это сбор статистики. В некоторых операционных системах предоставляется механизм построения гистограммы, показывающей положение счетчика команд программы пользователя. Таким образом, пользователь может видеть, какие процедуры его программы какой процент процессорного времени потребляют. Для этого на каждом тике драйвер часов должен проверить, собирается ли статистика по текущему процессу, и если да, то определяет, в каком диапазоне адресов находится счетчик команд. После этого значение счетчика, соответствующее этому диапазону, увеличивается на единицу. Такой же метод может применяться для получения статистики по самой операционной системе.

«Мягкие» таймеры

У большинства компьютеров есть второй программируемый таймер, который может быть установлен для формирования прерываний с той частотой, какая требуется программе. Этот таймер представляет собой добавление к основному системному таймеру, описанному в предыдущих разделах. До тех пор пока частота прерываний невелика, никаких проблем, связанных с использованием второго таймера для прикладных целей, не возникает. Трудности появляются, когда частота прерываний прикладного таймера становится очень высокой. Ниже мы кратко опишем схему программного таймера, хорошо работающую в различных обсто-

ятельствах, даже на высоких частотах. Идея обязана своим появлением Арону и Друшелю [15]. Подробности, пожалуйста, смотрите в этой статье.

Обычно есть два способа управления вводом-выводом: прерывания и опрос. Прерывания обладают низким временем задержки, то есть они происходят немедленно после самого события или с минимальной задержкой. С другой стороны, в современных центральных процессорах прерываниям сопутствуют значительные накладные расходы, связанные с необходимостью переключения контекста, а также с их влиянием на конвейер, кэш и буфер быстрого преобразования адреса TLB.

Вместо прерываний может использоваться опрос приложением какого-либо порта или слова памяти при ожидании события. Этот метод позволяет избежать прерываний, но может привести к значительным задержкам, то есть к замедленной реакции приложения на ожидаемое им событие. Это связано с тем, что событие может произойти сразу после опроса, в результате чего задержка реакции составит почти целый интервал опроса. В среднем задержка составит половину периода опроса.

Для некоторых приложений ни накладные расходы прерываний, ни задержка опроса неприемлемы. Возьмите, к примеру, такую высокоскоростную сеть, как гигабитная сеть Ethernet. Эта сеть способна принимать или доставлять пакет полного размера каждые 12 мкс. Для поддержания оптимальной производительности на выходе надо посылать новый пакет каждые 12 мкс.

Один из способов достижения такой скорости состоит в том, что по завершении передачи каждого пакета происходит прерывание, либо устанавливается таймер, инициирующий прерывания каждые 12 мкс. Недостаток этого метода — как показали измерения, для процессора Pentium II с частотой 300 МГц одно прерывание занимает 4,45 мкс (1335 тактов процессора) [15]. Этот показатель накладных расходов вряд ли улучшился с 70-х годов. Например, у большинства мини-компьютеров прерывание занимает всего четыре цикла шины, необходимых для помещения в стек счетчика команд и слова состояния процессора, и для загрузки новых значений PC и PSW. Сегодняшним процессорам приходится иметь дело с конвейером, MMU, TLB и кэшем, что увеличивает накладные расходы в несколько раз. Со временем эти эффекты только ухудшаются, не позволяя использовать прерывания от таймера с высокой частотой.

Идея «мягких» таймеров позволяет избежать лишних прерываний. Вместо этого ядро, вызываемое по какой-либо другой причине, перед тем как вернуться в режим пользователя, проверяет значение часов реального времени, чтобы проверить, не истек ли период ожидания «мягкого» таймера. Если время ожидания истекло, выполняется планируемое событие (например, передача пакета или проверка, не пришел ли пакет). При этом отпадает необходимость специального переключения в режим ядра, так как система и так уже находится в режиме ядра. Когда необходимые действия выполнены, «мягкий» таймер снова устанавливается для ожидания следующего события. Все, что для этого требуется — это взять текущее значение часов, прибавить к нему интервал ожидания и сохранить сумму в ячейке таймера.

«Мягкие» таймеры устанавливаются и срабатывают с той скоростью, с которой выполняются входы в ядро по другим причинам. К этим причинам относятся:

1. Системные вызовы.
2. Ошибки преобразования адреса TLB.

3. Отсутствие страницы памяти.
4. Прерывания ввода-вывода.
5. Временное отсутствие работы для центрального процессора.

Для определения частоты этих событий Арон и Друшель произвели измерения с несколькими вариантами загрузки центрального процессора, включая полностью загруженный web-сервер, выполняющий ограниченное скоростью вычислений фоновое задание, воспроизведение скачиваемого с Интернета аудио в режиме реального времени, а также перекомпиляцию ядра системы UNIX. Средний период обращений к ядру варьировался в диапазоне от 2 до 18 мкс. Примерно половину этих обращений составляли системные вызовы. Таким образом, в первом приближении вызов «мягкого» таймера через каждые 12 мкс является вполне выполнимым делом, хотя при этом иногда возможен пропуск временных сроков. Однако для таких приложений, как отсылка пакетов, лучше иногда опоздать с отправкой пакета на 10 мкс, чем затрачивать на прерывания до 35 % времени центрального процессора.

Конечно, могут быть периоды, когда нет системных вызовов, ошибок TLB или отсутствия страниц памяти. В этом случае «мягкий» таймер не будет обрабатываться и остановится. Для таких интервалов времени может быть принудительно установлена верхняя граница при помощи второго аппаратного таймера, срабатывающего, скажем, раз в 1 мс. Если приложению достаточно всего лишь 1000 пакетов в секунду, тогда комбинация «мягких» таймеров и низкочастотного аппаратного таймера может оказаться лучше, чем ввод-вывод, основанный только на прерываниях или только на опросе.

Алфавитно-цифровые терминалы

У каждого универсального компьютера есть по крайней мере одна клавиатура и один дисплей (монитор или плоский экран), используемые для общения с компьютером. Хотя клавиатура и дисплей персонального компьютера технически являются отдельными устройствами, они сообща образуют пользовательский интерфейс. К мэйнфреймам часто присоединяются специальные устройства, состоящие из клавиатуры и дисплея, за которыми могут работать удаленные пользователи. Такие устройства исторически называются **терминалами**. Мы будем продолжать использовать этот термин даже при обсуждении персональных компьютеров (по большей части из-за отсутствия лучшего термина).

Существует много разновидностей терминалов. На практике сегодня наиболее часто встречаются следующие три типа терминалов.

1. Автономные терминалы с последовательным интерфейсом RS-232 для связи с мэйнфреймами.
2. Дисплеи персональных компьютеров с графическим интерфейсом пользователя.
3. Сетевые терминалы.

Каждый из этих типов терминалов занимает свою «экологическую» нишу. В следующих разделах мы опишем все эти типы терминалов по очереди.

Технические средства терминалов с интерфейсом RS-232

Терминалы с интерфейсом RS-232 представляют собой технические устройства, состоящие из клавиатуры и дисплея, общающиеся по последовательному интерфейсу (рис. 5.30). Эти терминалы соединяются с интерфейсной платой при помощи 9-контактного или 25-контактного разъема. Один из контактов разъема используется для передачи данных, другой контакт — для получения данных, еще один контакт представляет собой заземление. Остальные контакты используются для различных управляющих функций, большая часть которых не используется. Линии, по которым символы посылаются побитно (в противоположность передаче стразу по 8 бит параллельно, как обычно соединяются с персональными компьютерами принтеры), называются **линиями последовательной передачи**. Этот интерфейс также используется всеми модемами. В системе UNIX линии последовательной передачи имеют имена вроде */dev/tty1* или */dev/tty2*. В системе Windows они обычно называются *COM1* и *COM2*.

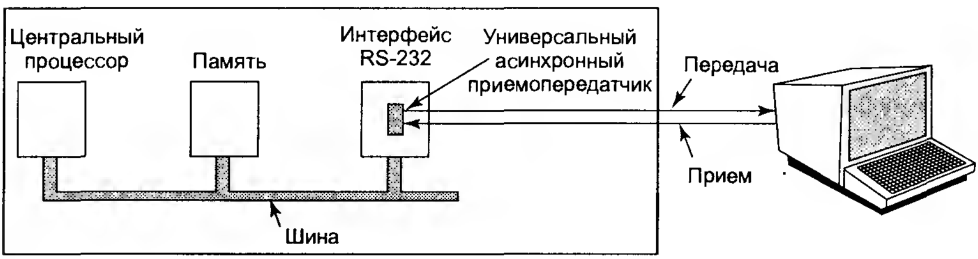


Рис. 5.30. Терминал с интерфейсом RS-232 общается с компьютером побитно

Чтобы послать символ по линии последовательной передачи на терминал с интерфейсом RS-232 или модем, компьютер должен передавать данные по одному биту, начиная передачу каждого символа со стартового бита и заканчивая одним или двумя стоповыми битами для разделения символов. Перед стоповыми битами может также добавляться бит четности, обеспечивающий рудиментарное обнаружение ошибок, что обычно требуется только при связи с мэйнфреймами.

Терминалы с интерфейсом RS-232 все еще применяются на мэйнфреймах, иногда соединенные по телефонной линии через модем. Их можно встретить в аэропортах, банках и других организациях. Даже когда они заменяются персональными компьютерами, эти персональные компьютеры часто просто эмулируют старые терминалы с интерфейсом RS-232, чтобы не менять программное обеспечение мэйнфреймов.

Такие терминалы также доминировали в мире мини-компьютеров. Большая часть программного обеспечения, созданного в тот период, основывалось на этих терминалах. Например, этот тип устройств поддерживается всеми системами UNIX.

Однако, что еще важнее, многие современные системы UNIX (а также и другие операционные системы) предоставляют возможность создать окно, состоящее из текстовых строк. Многие программисты работают практически исключительно

в текстовом режиме в таких окнах, даже на персональных компьютерах или рабочих станциях. Эти окна обычно эмулируют терминалы с интерфейсом RS-232 (либо ANSI-стандарт терминала этого типа), поэтому в них может работать огромное количество программ, написанных для подобных терминалов. За долгие годы это программное обеспечение, например текстовые редакторы *vi* и *emacs*, было полностью очищено от ошибок и обладает исключительной надежностью, свойством, чрезвычайно ценимым программистами.

Программное обеспечение, работающее с клавиатурой и дисплеем для этих окон, эмулирующих терминал, ничем не отличается от программного обеспечения, использующегося для настоящих терминалов. Поскольку эмуляторы этих терминалов пользуются такой широкой популярностью, программное обеспечение для них сохраняет свое значение, поэтому мы опишем его в следующих двух разделах.

Терминалы с интерфейсом RS-232 являются алфавитно-цифровыми терминалами. Это означает, что экран или окно отображает определенное количество строк текста. Обычный размер такого окна составляет 25 строк по 80 символов. Хотя иногда такие терминалы (и эмуляторы) поддерживают определенные специальные символы, в основном они являются исключительно текстовыми.

Поскольку как компьютеры, так и терминалы работают с целыми символами, но вынуждены обмениваться битами по последовательной линии, были разработаны специальные микросхемы для выполнения преобразований символов в последовательность битов и обратно. Они называются **универсальными асинхронными приемопередатчиками** (UART, Universal asynchronous receiver/transmitter). Микросхемы UART монтируются на интерфейсных картах, вставляемых в разъем шины компьютера, как показано на рис. 5.30. На многих компьютерах один или два последовательных порта встроены в материнскую плату.

Чтобы вывести символ на экран, драйвер терминала записывает этот символ в интерфейсную карту, в которой она буферизируется, после чего поразрядно выдвигается в последовательную линию универсальным асинхронным приемопередатчиком. Например, для аналогового модема, работающего со скоростью 56 000 бит/с, для передачи одного символа требуется немного более 179 мкс. Поскольку такая скорость передачи низка, драйвер обычно передает один символ в интерфейсную карту RS-232. После этого драйвер блокируется и ждет прерывания, которое инициирует интерфейс, передав символ и перейдя в состояние готовности к приему следующего символа. Микросхема UART способна одновременно передавать и принимать символы. Прерывание также генерируется при получении символа, и обычно несколько принятых символов могут сохраняться в буфере. Получив прерывание, драйвер терминала должен проверить регистр, чтобы определить причину прерывания. Некоторые интерфейсные карты имеют собственный процессор и память и могут одновременно поддерживать несколько линий, разгружая тем самым центральный процессор.

Терминалы с интерфейсом RS-232 могут быть разделены на три категории. Наиболее простыми являются печатающие терминалы или телетайпы. Символы, набираемые на клавиатуре, посылаются компьютеру. Символы, посланные компьютером, печатаются на бумаге. Такие терминалы уже давно считаются устаревшими и почти не встречаются, разве только в качестве примитивных принтеров.

Примитивные электронно-лучевые терминалы работают похоже, но вместо бумаги они выводят символы на экран. Их также называют «стеклянными телетайпами» (glass ttys), поскольку функционально они аналогичны печатающим телетайпам. Термин «tty» является сокращением слова Teletype®, означающего имя компании, бывшей пионером в области компьютерных терминалов. Теперь сокращение «tty» используется для обозначения любого терминала. Стеклянные телетайпы также устарели.

Умные электронно-лучевые терминалы на самом деле представляют собой небольшие специализированные компьютеры. У них есть процессор и память. Они также содержат программное обеспечение, хранящееся, как правило, в ПЗУ. С точки зрения операционной системы основное различие между стеклянным телетайпом и умным терминалом состоит в том, что последний понимает управляющие последовательности символов, называемые ESC-последовательностями. При помощи передачи такому терминалу символа ASCII ESC (0x1B), за которым передается еще несколько других символов, можно управлять выводом на экран терминала. Например, с помощью ESC-последовательности можно переместить курсор на новую позицию, вывести текст в любое заданное место экрана, очистить экран и т. д. Именно такие терминалы используются в системах мэйнфреймов и эмулируются другими операционными системами. Ниже мы обсудим программное обеспечение умных терминалов.

Программное обеспечение ввода

Клавиатура и дисплей являются почти независимыми устройствами, поэтому мы будем рассматривать их здесь по отдельности. Однако они не совсем независимы, так как вводимый с клавиатуры символ обычно выводится на экран.

Основная работа клавиатурного драйвера состоит в сборе ввода с клавиатуры и передаче его программам, читающим с терминала. Существует две философские концепции, описывающие работу драйвера. Согласно первой концепции, работа драйвера заключается в сборе ввода и передаче его программам безо всяких изменений. Программа, читающая с терминала, получает необработанные последовательности ASCII-символов. (Передавать программам пользователя коды клавиш неприемлемо, так как они в большой степени зависят от конкретной машины.)

Эта философия хорошо удовлетворяет потребности таких сложных текстовых редакторов, как *emacs*, который позволяет пользователю связать любое действие с любым символом или последовательностью символов. Однако это означает, что если пользователь вместо *date* наберет на клавиатуре *dste*, а затем исправит ошибку, удалив три последние символа и допечатав символы *ate*, за которыми нажмет Enter, программа пользователя получит 11 следующих ASCII-символов:

```
dste←←←ateCR
```

Не всем программам нужны эти подробности. Чаще всего им нужна уже исправленная строка, а не вся последовательность введенных символов. Таким образом, формируется вторая философская концепция: драйвер выполняет все редактирование внутри строки, а программе пользователя передает уже исправленную строку. Первая философская концепция является символьно-ориентированной, вторая —

строчно-ориентированной. Изначально эти режимы работы драйвера назывались **режимом без обработки** (или «сырым» режимом) и **режимом с обработкой**. В стандарте POSIX режим с обработкой называется **каноническим режимом**. **Неканонический режим** соответствует режиму без обработки, хотя многие детали поведения терминала могут различаться. Совместимые со стандартом POSIX системы предоставляют несколько библиотечных функций, поддерживающих выбор любого из этих двух режимов, а также изменение многих аспектов конфигурации терминала.

Основная задача клавиатурного драйвера состоит в сборе символов. Если каждое нажатие на клавишу вызывает прерывание, драйвер может получать введенный символ во время обработки прерывания. Если прерывания преобразуются низкоуровневым программным обеспечением в сообщения, каждый полученный символ может помещаться в сообщение. В качестве альтернативы символ может помещаться в небольшой буфер в памяти, а сообщение использовать только для извещения драйвера о том, что что-то прибыло. Второй подход более надежен, особенно если сообщение может быть послано только ожидающему его процессу, а драйвер клавиатуры может оказаться занятым обработкой предыдущего символа.

Если терминал находится в каноническом режиме (режиме с обработкой), введенные символы должны храниться в буфере до тех пор, пока не будет введена полная строка. Даже если терминал находится в «сыром» режиме, может оказаться, что программа еще не запрашивала входные данные, поэтому введенные символы все равно должны буферизироваться, чтобы позволить пользователю производить упреждающий ввод. (Разработчиков систем, не позволяющих пользователям вводить символы с клавиатуры заранее, следует обмазывать дегтем и вываливать в перьях, так как заставлять их пользоваться собственной системой было бы слишком жестоким наказанием.)

Для буферизации символов обычно применяются два метода. В первом случае в драйвере содержится центральный пул буферов, в каждом из которых хранится около 10 символов. С каждым терминалом связана структура данных, содержащая, среди прочего, указатель на цепочку буферов, в которых находятся символы, введенные с данного терминала. Чем больше символов введено, тем больше выделяется буферов, соединенных в цепь. Когда символ передается программе пользователя, буферы удаляются и память возвращается центральному пулу.

Другой подход состоит в том, что буферизация производится прямо в структуре данных терминала, без центрального пула буферов. Поскольку пользователи часто печатают команду, обработка которой требует некоторого времени (например, перекомпиляция и сборка большой двоичной программы), а затем печатают еще несколько строк, буфер драйвера должен вмещать не меньше 200 символов для каждого терминала. В крупной системе разделения времени с 100 терминалами постоянное выделение 20 Кбайт на буфер ввода с клавиатур кажется чрезмерным, поэтому центральный пул буферов размера около 5 Кбайт будет, видимо, достаточным. С другой стороны, при выделенном буфере для отдельного терминала драйвер становится проще (не требуется управления связанным списком). Такой подход является предпочтительным на персональном компьютере с единственной клавиатурой. Рисунок 5.31 иллюстрирует разницу между этими двумя методами.

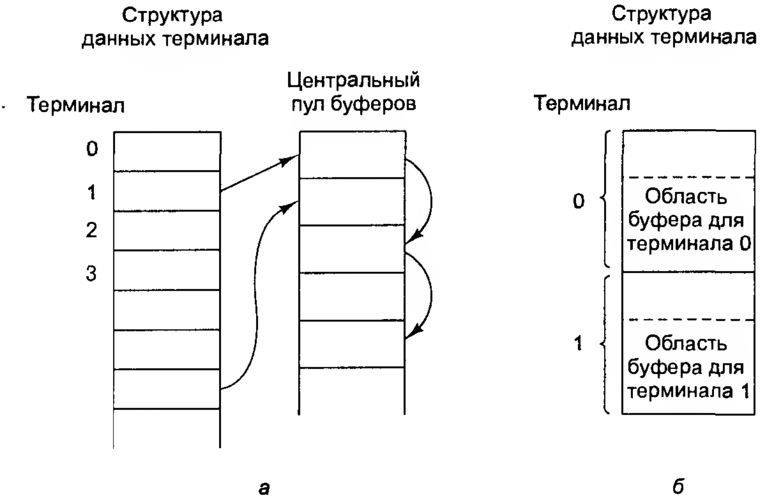


Рис. 5.31. Центральный пул буферов (*а*); выделенный буфер для каждого терминала (*б*)

Хотя клавиатура и дисплей являются логически раздельными устройствами, многие пользователи привыкли видеть только что введенные с клавиатуры символы отображаемыми на экране. Некоторые (старые) терминалы должны были автоматически (аппаратно) отображать все, что вводилось с клавиатуры, что не только крайне неудобно при вводе паролей, но также значительно ограничивает гибкость сложных редакторов и других программ. К счастью, на большинстве терминалов при нажатии клавиши ничего автоматически не отображается. Отображением символов на экране занимается исключительно программное обеспечение. Этот процесс называется **печатью эха** или **эхопечатью**.

Печать эха усложняется тем фактом, что во время нажатия пользователем клавиши программа может осуществлять вывод на экран. По меньшей мере, драйвер должен решить, где поместить эхо так, чтобы оно не исчезло под выводом программы[1].

Кроме того, если пользователь ввел более 80 символов в одной строке, вывод эха на 80-символьном экране может осуществляться по-разному. В зависимости от приложения переход на следующую строку может оказаться приемлемым либо неприемлемым. Некоторые драйверы просто усекают все введенные строки до 80 символов, игнорируя все символы после 80-й колонки.

Еще одна проблема заключается в обработке табуляторов. Обычно драйвер вычисляет текущую позицию курсора, учитывая как вывод программы, так и вывод эха ввода, после чего вычисляет число отображаемых вместо табулятора пробелов[2].

---

[1] Точнее говоря, это проблема не ввода, то есть драйвера клавиатуры, а вывода. Собственно, и проблемы никакой нет. Все символы просто поступают в общий поток стандартного вывода. — *Примеч. перев.*

[2] Драйвер клавиатуры тут, скорее всего, ни при чем. Преобразованием табулятора в пробелы должна заниматься либо программа более высокого уровня, чем драйверы, либо, в крайнем случае, драйвер вывода. — *Примеч. перев.*

Наконец, существует проблема эквивалентности устройств. Логически в конце строки текста требуется символ возврата каретки, чтобы переместить курсор обратно к колонке 1, и символ перевода строки для перемещения курсора на следующую строку. Требовать от пользователя вводить оба символа — вряд ли удачная мысль, хотя на некоторых терминалах имеется специальная клавиша, посылающая оба символа с 50-процентной вероятностью сделать это именно в том порядке, в котором их ожидает программа. Преобразование всего, что поступает с клавиатуры в стандартный внутренний формат, используемый операционной системой, является одной из задач драйвера.

Если стандартом предусматривается хранение только символов перевода строки (соглашение UNIX), тогда символы возврата каретки должны преобразовываться в символы перевода строки. Если внутренний формат предусматривает хранение обоих символов (соглашение Windows), тогда драйвер должен формировать символ перевода строки при получении символа возврата каретки и символ возврата каретки при получении символа перевода строки. Независимо от внутренних соглашений, терминал может требовать вывода обоих символов для корректного управления выводом на экран. Поскольку к большому компьютеру могут оказаться подключенными терминалы различных типов, драйвер клавиатуры должен заниматься преобразованием всех различных комбинаций символа возврата каретки и символа перевода строки во внутренний стандарт, а также следить за правильностью эхопечати.

При работе в каноническом режиме некоторые вводимые символы имеют особое значение. В табл. 5.4. показаны все специальные символы, требуемые стандартом POSIX. По умолчанию все они являются управляющими символами, которые не должны конфликтовать с вводимым текстом или кодами, используемыми программами. Однако все символы, кроме последних двух, могут быть программно изменены.

Таблица 5.4. Специальные символы канонического режима

Символ	Имя в POSIX	Комментврий
CTRL+H	ERASE	Удалить один символ слева
CTRL+U	KILL	Удалить всю введенную строку
CTRL+V	LNEXT	Интерпретировать следующий символ буквально
CTRL+S	STOP	Остановить вывод
CTRL+Q	START	Начать вывод
DEL	INTR	Прервать процесс (SIGINT)
CTRL+\	QUIT	Форсировать дамп памяти (SIGQUIT)
CTRL+D	EOF	Конец файла
CTRL+M	CR	Возврат каретки (неизменный)
CTRL+J	NL	Перевод строки (неизменный)

Символ *ERASE* позволяет пользователю удалить один только что введенный символ. Обычно для этого применяется клавиша «забой» (backspace) или комбинация клавиш CTRL+H (обоим вариантам соответствует код 0x08). Этот символ не добавляется к очереди символов, а, наоборот, удаляет предыдущий символ из

очереди. Печать эха для такого символа должна выглядеть как последовательность трех символов: перемещение курсора на одну позицию влево, пробел и еще раз возврат на одну позицию, чтобы удалить с экрана предыдущий символ. Если же предыдущим символом был табулятор, его удаление зависит от того, как табулятор был отработан при печати. Если он был преобразован в пробелы, то необходима дополнительная информация о том, насколько далеко следует возвращать курсор. Если же сам табулятор хранится в очереди ввода, он может быть удален, а вся строка напечатана еще раз. В большинстве систем символ *ERASE* удаляет символы текущей строки. Символы предыдущей строки и разделяющие строки символы возврата каретки или перевода строки не удаляются.

Если пользователь обнаруживал ошибку в начале введенной строки, то единственным способом ее исправления во многих старых системах, не позволяющих перемещать курсор по строке, являлось полное удаление всей строки. В этом случае бывало удобнее воспользоваться специальным символом *KILL*, удалявшим всю строку сразу. В некоторых системах эта строка полностью исчезала с экрана, но в некоторых она оставалась на экране, включая возврат каретки и перевод строки, так как многие пользователи любят видеть свою старую строку. Как и символ *ERASE*, символ *KILL* работает только с текущей строкой. При удалении блока символов драйвер может вернуть освободившиеся буферы в пул.

Иногда символы *ERASE* или *KILL* должны быть введены в строку как обычные данные. Для этого служит символ *LNEXT*, действующий в качестве **префиксного символа**[1]. В системе UNIX для этого по умолчанию используется сочетание клавиш CTRL+V (код 0x16). В более старых системах UNIX в качестве символа *KILL* часто используется символ @, но впоследствии этот символ стал использоваться в адресах электронной почты сети Интернет, как, например, linda@cs.washington.edu. Те, кому привычнее старые соглашения, могут переопределить символ *KILL* как @, но тогда им придется вводить символ @ буквально при вводе адреса электронной почты. Это можно сделать, нажав на клавиатуре последовательно клавиши CTRL+V и @. Сам символ *LNEXT* может быть введен, если дважды нажать клавиши CTRL+V. Встретив символ *LNEXT*, драйвер установит флаг, означающий, что следующий символ не следует подвергать специальной обработке. Сам символ *LNEXT* не устанавливается в очередь символов.

Чтобы приостановить и продолжить вывод на экран, также предоставляются специальные управляющие коды. В UNIX это символы *STOP* (CTRL+S) и *START* (CTRL+Q). Эти символы не хранятся в буфере, но используются для установки и сброса флага в структуре данных терминала. При каждой операции вывода на экран проверяется значение этого флага. Если флаг установлен, вывод не производится. Эхо при этом обычно также подавляется.

Часто возникает необходимость прервать выполнение отлаживаемой программы. Для этой цели могут использоваться символы *INTR* (DEL) и *QUIT* (CTRL+\). В системе UNIX клавиша DEL посылает сигнал прерывания SIGINT всем процес-

---

[1] В англоязычной литературе префиксный символ часто называется ESC-символом, так же как и различные управляющие последовательности ESC-последовательностями. Хотя в качестве префиксного в разных ситуациях могут использоваться различные символы. Строго говоря, ESC-последовательность — это последовательность символов, начинающаяся с символа ESC, то есть символа ASCII 0x1B. — *Примеч. перев.*

сам, запущенным с этого терминала. Реализация может быть непростой. Наиболее сложным является передача информации от драйвера в ту часть системы, которая занимается обработкой сигналов, поскольку она не ожидает получения подобной информации. Результат нажатия клавиш CTRL+\ (код 0x1C) аналогичен нажатию клавиши DEL, с той разницей, что процессам посылается сигнал SIGQUIT, вызывающий прекращение работы процесса с сохранением дампа памяти, если этот сигнал специально не перехватывается процессом. При нажатии любой из этих клавиш драйвер должен вывести эхо в виде перевода строки и возврата каретки, а также очистить свой буфер с накопленными введенными символами, чтобы позволить начать новый ввод. Часто вместо клавиши DEL для символа *INTR* по умолчанию используется сочетание клавиш CTRL+C (код 0x03), так как с появлением электронно-лучевых дисплеев многие программы стали использовать клавишу DEL для удаления символа справа от курсора при редактировании.

Специальный символ *EOF* (CTRL+D), означающий конец файла в системе UNIX, сообщает ожидающей ввода программе, что информации на входе больше не будет. Программа действует так, как если бы при чтении из файла достигла его конца.

Некоторые драйверы терминала предоставляют возможность более сложного редактирования строки, чем было описано здесь. Они имеют специальные управляющие символы, позволяющие удалять целиком слова, перемещать курсор вперед и назад по символам и по словам, вставлять текст в середину уже набранной строки и т. д. Добавление подобных функций к драйверу значительно увеличивает его. Кроме того, эти функции чаще всего оказываются неиспользуемыми экранными редакторами, предпочитающими работать с драйверами клавиатуры в «сыром» режиме.

Программное обеспечение вывода

Терминальный вывод несколько проще ввода. По большей части компьютер посылает символы терминалу, который их отображает. Обычно блок символов, например строка, записывается на терминал за один системный вызов. Как правило, метод, используемый для терминалов с интерфейсом RS-232, состоит в том, что для каждого терминала выделяется выходной буфер. Эти буферы могут входить в тот же пул буферов, что и входные буферы, или представлять собой выделенные буферы. Вывод эха на терминал также копируется в буфер. После того как символы помещены в выходной буфер, первый символ выводится на терминал, после чего драйвер блокируется. Когда приходит прерывание, извещающее драйвер о готовности терминала принять следующий символ, посылается следующий символ и т. д.

Экранным редакторам и другим сложным программам бывает нужно перерисовать экран, заменив определенные участки экрана и не меняя остального текста. Для этого многие терминалы поддерживают наборы управляющих команд, позволяющие перемещать курсор, удалять строки и т. д. Эти команды часто реализуются в виде **ESC-последовательностей**, то есть последовательностей символов, начинающихся с символа ESC (0x1B). Во времена расцвета терминалов с интерфейсом RS-232 существовали сотни разновидностей терминалов, у каждого из которых был свой набор ESC-последовательностей. В результате было довольно

сложно написать программное обеспечение, работающее более чем на одном типе терминалов.

В системе Berkley UNIX было предложено решение этой проблемы, заключающееся в базе данных терминалов и называющееся **termcap**. Этот программный пакет определял множество основных действий, таких как перемещение курсора на нужную колонку и строку. Чтобы переместить курсор в определенное место, программа, например текстовый редактор, формировала свою ESC-последовательность, которая преобразовывалась в ESC-последовательность, соответствующую тому конкретному терминалу, на который производился вывод. Таким образом, текстовый редактор мог работать на любом терминале, включенном в базу данных termcap.

В конце концов производители компьютеров и программного обеспечения осознали необходимость стандартизации ESC-последовательностей, в результате чего был разработан стандарт ANSI. Некоторые примеры ESC-последовательностей этого стандарта приведены в табл. 5.5.

Таблица 5.5. Некоторые ESC-последовательности стандарта ANSI

ESC-последовательность	Значение
ESC [*n*A	Переместить курсор вверх на *n* строк
ESC [*n*B	Переместить курсор вниз на *n* строк
ESC [*n*C	Переместить курсор вправо на *n* позиций
ESC [*n*D	Переместить курсор влево на *n* позиций
ESC [*m*;*n*H	Переместить курсор в позицию (*m*, *n*)
ESC [*s*J	Очистить экран от курсора (0 до конца, 1 от начала, 2 весь)
ESC [*s*K	Очистить строку от курсора (0 до конца, 1 от начала, 2 всю)
ESC [*n*L	Вставить *n* строк у курсора
ESC [*n*M	Удалить *n* строк у курсора
ESC [*n*P	Удалить *n* символов у курсора
ESC [*n*@	Вставить *n* символов у курсора
ESC [*n*m	Разрешить выделение текста (0 — нормальный, 4 — полужирный, 5 — мерцающий, 7 — инверсный)
ESC M	Скроллинг экрана в обратную сторону, если курсор находится в верхней строке

Рассмотрим, как эти ESC-последовательности могут использоваться текстовым редактором. Предположим, пользователь дает редактору команду удалить строку 3, а затем закрыть промежуток между строками 2 и 4. Для этого редактор может послать терминалу по последовательной линии следующую ESC-последовательность:

```
ESC [3:1H ESC [0K ESC [1M
```

(где пробелы используются только для разделения символов и не передаются в линию). Эти последовательности перемещают курсор на начало строки 3, удаляют всю строку, а затем сдвигают строки, начиная с четвертой, вверх на одну позицию. Аналогичные ESC-последовательности могут использоваться для вставки в середину текста. Подобным же образом добавляются и удаляются слова.

Графические интерфейсы пользователя

На персональных компьютерах могут использоваться символьные интерфейсы. В течение нескольких лет доминировала система MS-DOS с символьным интерфейсом. Однако теперь на большинстве персональных компьютеров используется **графический интерфейс пользователя** (GUI, Graphical User Interface). Сокращение GUI произносится как «гуи» («gooey»).

Графический интерфейс пользователя был придуман Дугласом Энгельбартом и его исследовательской группой в Стенфордском исследовательском институте. Затем этот интерфейс был скопирован исследователями из Xerox PARC. Однажды Стив Джобс, один из основателей компании Apple, посетив PARC, увидел графический интерфейс пользователя на компьютере Xerox. Это натолкнуло его на мысль о создании нового компьютера, которым стал компьютер Lisa фирмы Apple, появившийся в 1983 году. Lisa была слишком дорогой машиной и поэтому она не получила коммерческого успеха, но ее преемник Macintosh, разработанный годом позже, стал крайне популярен. Компьютер Apple Macintosh оказал значительное влияние на систему Windows, первая версия которой была представлена корпорацией Microsoft в 1985 году, а также на другие системы с графическим интерфейсом пользователя.

Графический интерфейс пользователя состоит из четырех основных элементов, из первых букв английских названий которых можно сложить слово WIMP (Windows, Icons, Menus, Pointing device — окна, пиктограммы, меню, указывающее устройство). Окна представляют собой прямоугольные участки экрана, используемые для запуска программ. Пиктограммами называются небольшие символы, на которых можно щелкнуть мышью, чтобы вызвать какое-либо действие. Меню являются списками действий, из которых может быть выбрано одно. Наконец, указывающее устройство — это мышь, шаровой манипулятор или другое устройство, используемое для перемещения курсора по экрану и для выбора элементов.

Программное обеспечение графического интерфейса пользователя может быть реализовано либо на уровне пользователя, как это делается в семействе систем UNIX, либо включено в саму операционную систему, как в случае Windows. В следующих разделах мы познакомимся с аппаратурой и программным обеспечением ввода и вывода для графического интерфейса пользователя персональных компьютеров. В первую очередь будет обсуждаться операционная система Windows, однако основные понятия действительны и для других графических интерфейсов пользователя.

Аппаратное обеспечение клавиатуры, мыши и дисплея персонального компьютера

Все современные персональные компьютеры оснащены клавиатурами и растровым графическим дисплеем с изображением, отображаемым в памяти компьютера. Эти компоненты составляют части самого компьютера. Однако в современном персональном компьютере клавиатура и экран являются полностью раздельными устройствами, каждое со своим собственным драйвером.

Связь с клавиатурой может осуществляться через последовательный порт, параллельный порт или порт USB. При нажатии любой клавиши центральный процессор прерывается, и драйвер клавиатуры извлекает символ, читая порт ввода-вывода. Все остальное осуществляется программно, в основном в драйвере клавиатуры.

На Pentium-компьютерах клавиатура содержит встроенный микропроцессор, общающийся с микросхемой контроллера, расположенной на материнской плате, через специальный последовательный порт. Прерывание возникает при каждом нажатии, а также при каждом отпускании клавиши. Аппаратная часть клавиатуры поставляет в порт не ASCII-код клавиши, а ее номер, или, как его еще называют, скан-код. Например, при нажатии клавиши A в регистр ввода-вывода помещается код клавиши 30. При этом драйвер должен решить, был ли этот символ строчным, прописным или частью какой-либо комбинации клавиш вроде CTRL+A, ALT+A, CTRL+ALT+A и т. д. Для этого драйвер должен запоминать все нажатые, но еще не отпущенные клавиши (например, SHIFT).

Например, последовательность действий

```
Нажать SHIFT, нажать A, отпустить A, отпустить SHIFT
```

означает прописной символ А. Однако последовательность действий

```
Нажать SHIFT, нажать A, отпустить SHIFT, отпустить A
```

также означает прописной символ А. Поскольку клавиатурный интерфейс возлагает всю тяжесть обработки на программное обеспечение, он является исключительно гибким. Например, программе пользователя может быть небезразлично, введена ли цифра с верхней линейки клавиш или с правой цифровой клавиатуры. В принципе драйвер может предоставить такую информацию.

У большинства персональных компьютеров имеется мышь или, в некоторых случаях, шаровой манипулятор, представляющий собой обычную мышь, лежащую на спине. Наиболее распространенный тип компьютерных мышей содержит в себе резиновый шар, выглядывающий из отверстия в днище мыши и вращающийся, когда мышь двигается по столу или коврику. При вращении шар поворачивает прижатые к нему резиновые ролики, закрепленные на перпендикулярных осях. При движении мыши с запада на восток вращается ось x, а движение мыши с севера на юг заставляет вращаться ось y. Когда мышь преодолевает по столу определенное минимальное расстояние или нажимается или отпускается одна из кнопок мыши, мышь посылает компьютеру сообщение. Обычно это минимальное расстояние, которое некоторые люди называют «мики», составляет около 0,1 мм. У мышей может быть одна, две или три кнопки, в зависимости от оценки разработчиками интеллектуальных способностей пользователей — смогут ли они работать более чем с одной клавишей.

Сообщение, посылаемое компьютеру, содержит три параметра: изменения позиции по координатам x и y, Δx и Δy и состояние кнопок. Формат сообщения зависит от системы и числа кнопок мыши. Обычно оно занимает 3 байт. Большинство мышей способно передавать до 40 сообщений в секунду, поэтому может оказаться, что с момента последнего сообщения мышь переместилась на несколько мики.

Обратите внимание, что мышь сообщает только об изменениях своей позиции, а не об абсолютном значении позиции. Если мышь аккуратно поднять и снова положить, так что шарик не повернется, то сообщений послано не будет.

Некоторые графические интерфейсы пользователя отличают однократный щелчок мыши от двойного щелчка. Если два щелчка мыши достаточно близки в пространстве[1] (в миках) и во времени (доли секунды), операционная система сигнализирует о двойном щелчке мыши. Временной интервал, отличающий двойной щелчок от двух независимых щелчков, а также скорость перемещения курсора мыши по экрану могут быть программно настроены пользователем.

Рассмотрим теперь аппаратную часть дисплея. Дисплеи могут быть разделены на две основные категории: устройства с **векторной графикой** и устройства с **растровой графикой**. Векторные графические устройства могут выполнять такие команды, как вывод точек, рисование линий, геометрических фигур и текста. У растровых графических устройств, напротив, область вывода представляет собой прямоугольную сетку точек, называемых **пикселами**, каждая из которых может принимать различные значения яркости или цвета. В ранние дни эпохи компьютеростроения векторные графические устройства встречались довольно часто, но в наши дни единственными векторными графическими устройствами остались плоттеры. Все остальные графические устройства используют растровую графику.

Растровые графические дисплеи реализуются при помощи специального устройства, называемого **графическим адаптером**. Графический адаптер содержит специальную память, называемую **видео-ОЗУ** или **видеопамятью** и образующую часть адресного пространства компьютера. Это означает, что центральный процессор обращается к ней так же, как и к остальной оперативной памяти (рис. 5.32). Здесь хранится образ экрана либо в символьном, либо в растровом виде. В символьном виде каждый байт (или два байта) видеопамяти содержат один отображаемый символ. В растровом виде каждый пиксел экрана представляется в видеопамяти отдельно. На каждый пиксел отводится от одного бита для простейшего бинарного черно-белого изображения до 24 бит для цветного дисплея высокого качества.

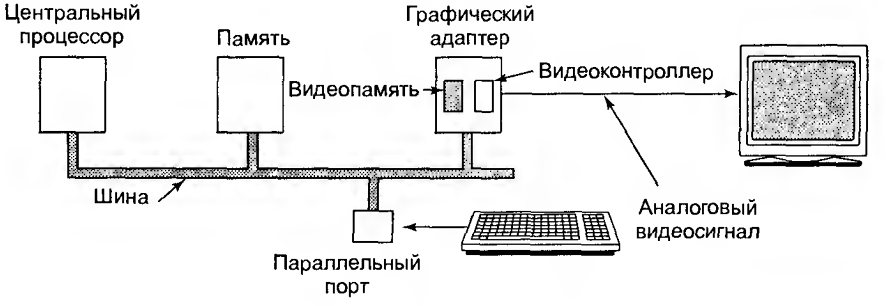


Рис. 5.32. Дисплей с общей памятью

Кроме того, в состав графического адаптера входит микросхема, называющаяся **видеоконтроллером**. Эта микросхема получает символы из видеопамяти и форми-

[1] Пространственный интервал без учета времени не имеет смысла и программно нереализуем. Достаточно использования только временного интервала, который, кстати, только и является настраиваемым, например, в системе Windows. — *Примеч. перев.*

рует соответствующий им видеосигнал, посылаемый на монитор. Монитор формирует луч электронов, сканирующий экран в горизонтальном направлении. Обычно на экран выводится от 480 до 1024 горизонтальных линий с количеством точек на линию от 640 до 1200. Сигнал видеоконтроллера модулирует электронный луч, определяя яркость каждого пиксела. У цветных мониторов три луча, для красного, зеленого и синего цветов, модулируемые независимо друг от друга. В плоских мониторах также используются пикселы трех цветов, но принципы работы этих мониторов выходят за рамки данной книги.

Видеоконтроллеры могут работать в двух режимах: символьном (используемом для простого текста) и растровом (для всего остального). В символьном виде контроллер преобразует каждый символ в прямоугольник пикселов размером 9×14 (включая промежутки между символами) и составляет из них экран из 25 строк по 80 символов. Для этого дисплей должен иметь 350 линий по 720 пикселов в каждой. Чтобы избежать мерцания, каждый кадр должен перерисовываться от 60 до 100 раз в секунду.

Для вывода текста на экран видеоконтроллер должен взять из видеопамяти первые 80 символов, сформировать для них 14 горизонтальных линий, которые подать на монитор, затем взять следующие 80 символов и т. д. Контроллер может также брать из видеопамяти по одному символу, что позволяет обойтись без буферизации символов в контроллере. Растры шрифтов 9×14 бит хранятся в ПЗУ, используемым видеоконтроллером. (ОЗУ может также использоваться для поддержки настраиваемых шрифтов.) Адресация обращений к ПЗУ 12-разрядная, 8 бит для кода символа и 4 бит для линии развертки. Восемь бит в каждом байте ПЗУ управляют 8 пикселами, 9-й пиксел между символами всегда пустой. Таким образом, для вывода строки текста на экран требуется 14 × 80 = 1120 обращений к памяти. Такое же число обращений требуется к знакогенератору в ПЗУ.

На рис. 5.33, *а* показана часть видеопамяти дисплея, работающего в символьном режиме. Каждый символ на экране (рис. 5.33, *б*) занимает в видеопамяти два байта. Младший байт слова содержит ASCII-код отображаемого символа. В старшем байте слова хранятся атрибуты символа, означающие его цвет, мерцание, инверсию и т. д. Для экрана из 25 строк по 80 символов требуется 4000 байт видеопамяти.

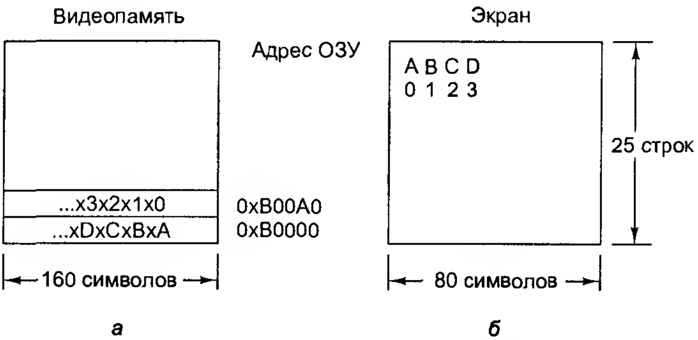


Рис. 5.33. Видеопамять монохромного дисплея, работающего в символьном виде (*а*); соответствующий экран (*б*). Символами x обозначены байты атрибутов

При работе дисплея в графическом режиме используются те же принципы, с той разницей, что каждый пиксел экрана управляется индивидуально и каждому пикселу соответствует своя область (от 1 до 24 бит) в видеопамяти. В простейшем случае бинарного изображения каждому пикселу экрана соответствует один бит видеопамяти. В случае высококачественного изображения на каждый пиксел приходится 24 бит видеопамяти, по 8 бит для интенсивности красного, зеленого и синего цветов. Представление цвета в виде разложения на составляющие интенсивности красного, зеленого и синего цветов, называющееся по первым буквам их английских названий RGB (Red, Green, Blue), обусловлено свойствами восприятия человеческого глаза.

Размеры современных растровых графических экранов варьируются в широких пределах. Самые распространенные стандарты: 640×480 (VGA), 800×600 (SVGA), 1024×768 (XGA), 1280×1024 и 1600×1200 пикселов. Все эти экраны, кроме 1280×1024, имеют соотношение размеров сторон 4:3, что соответствует стандартным телевизионным трубкам (NTSC или PAL/SECAM), и, следовательно, позволяет использовать квадратные пикселы. Размер 1280×1024 должен был на самом деле быть 1280×960, но привлекательность числа 1024, видимо, была слишком велика, чтобы противостоять этому искушению, несмотря на то что пикселы при этом слегка искажаются и преобразования в другие размеры усложняются. Цветному дисплею, работающему в режиме 1024×768 с 24 бит на пиксел, необходимо 2,25 Мбайт ОЗУ только для хранения изображения. Если весь экран обновляется 75 раз в секунду, видеопамять должна доставлять данные с постоянной скоростью 169 Мбайт/с.

Чтобы избежать необходимости управлять такими большими областями памяти, во многих системах имеется возможность использовать меньшее цветовое разрешение. В простейшей схеме каждый пиксел представляется 8-разрядным числом. Оно обычно не содержит самого цвета пиксела, а является индексом в таблице цветов, состоящей из 256 24-разрядных элементов формата RGB. Эта таблица, называемая **цветовой палитрой** и позволяющая экрану содержать в любой момент 256 произвольных цветов, часто реализуется аппаратно. При изменении, например, элемента 7 цветовой палитры, изменятся цвета всех пикселов, содержащих байт 7. Использование 8-разрядной цветовой палитры позволяет в три раза сократить размер, требуемый для хранения изображения, за счет более грубого цветового разрешения. Цветовая палитра также применяется в схеме сжатия изображений GIF (Graphics Interchange Format — формат графического обмена).

Также применяются цветовые палитры с 16 битами на пиксел. В этом случае цветовая палитра содержит 65 536 элементов, что позволяет одновременно использовать до 65 536 цветов. Такая схема позволяет достичь гораздо более качественной цветопередачи, однако размер требуемой видеопамяти при этом методе сокращается всего лишь на одну треть по сравнению с 24-разрядными пикселами. К тому же сама цветовая палитра размером 65 536 элементов по 3 байт (192 Кбайт) тоже должна где-то храниться. Если она хранится аппаратно (чтобы избежать затрат времени на дополнительные обращения к оперативной памяти), хранение такой палитры требует существенно больше аппаратных буферов памяти, чем в случае 8-разрядной цветовой палитры.

Возможно и хранение в 16-разрядных пикселах значений цвета в формате RGB с 5 бит на цвет, с одним битом лишним (можно выделить зеленой составляющей цвета 6 бит, так как человеческий глаз более чувствителен именно к зеленому цвету.) В результате может получиться система, схожая с 24-разрядным цветом, но с меньшим числом градаций яркости для каждого цвета.

Программное обеспечение ввода

Получив символ, клавиатурный драйвер должен начать его обработку. Поскольку программным обеспечением используются коды символов, а не скан-коды клавиш, получаемые драйвером от клавиатуры, драйвер должен преобразовать скан-коды в символы с помощью таблицы. Не все IBM-совместимые компьютеры используют стандартные скан-коды клавиш, поэтому, чтобы драйвер мог поддерживать различные клавиатуры, он должен осуществлять эти преобразования при помощи различных таблиц. Проще всего включить нужную таблицу в драйвер во время компиляции драйвера. Однако такой подход усложняется тем фактом, что огромному количеству пользователей требуется набирать тексты на языках, отличных от английского. В различных странах клавиатуры организуются по-разному, и даже в странах, использующих шрифты на основе латиницы, применяются различные акцентированные буквы, перечеркнутые буквы и т. п., а также знаки пунктуации, отсутствующие в английском языке.

Для достижения большей гибкости в настройке раскладок клавиатуры[1] многие операционные системы предоставляют загружаемые **кодовые страницы** или, как их еще называют, **карты клавиш**. Они позволяют выбирать способ преобразования скан-кодов клавиш в символы, предоставляемые приложению, либо во время загрузке системы, либо позднее.

Программное обеспечение вывода для Windows

Программное обеспечение вывода для графического интерфейса пользователя представляет собой весьма обширную тему. Только о графическом интерфейсе пользователя системы Windows написано множество книг по полторы тысячи страниц (например, [265, 303, 271]). Очевидно, в этом разделе мы сможем лишь поверхностно коснуться этой темы и познакомиться с некоторыми основными концепциями. Чтобы обсуждение было более конкретным, мы опишем программный интерфейс приложения Windows API, поддерживаемый всеми 32-разрядными версиями системы Windows. Программное обеспечение вывода для других графических интерфейсов пользователя значительно отличается в деталях, но в первом приближении имеет много общего с Windows API.

Базовым элементом любого графического интерфейса пользователя является прямоугольная область экрана, называемая **окном**. Положение окна и его размеры однозначно определяются координатами (в пикселах) двух противоположных углов окна. Окно может содержать заголовок, меню, вертикальную и горизонталь-

[1] А также чтобы избавить пользователя от необходимости транслировать и переустанавливать драйвер клавиатуры. — *Примеч. перев.*

ную полосы прокрутки. Типичное окно показано на рис. 5.34. Обратите внимание, что система координат, принятая в Windows, помещает начало координат в левый верхний угол, а координата *y* увеличивается сверху вниз, что отличается от картезианской системы координат, принятой в математике.

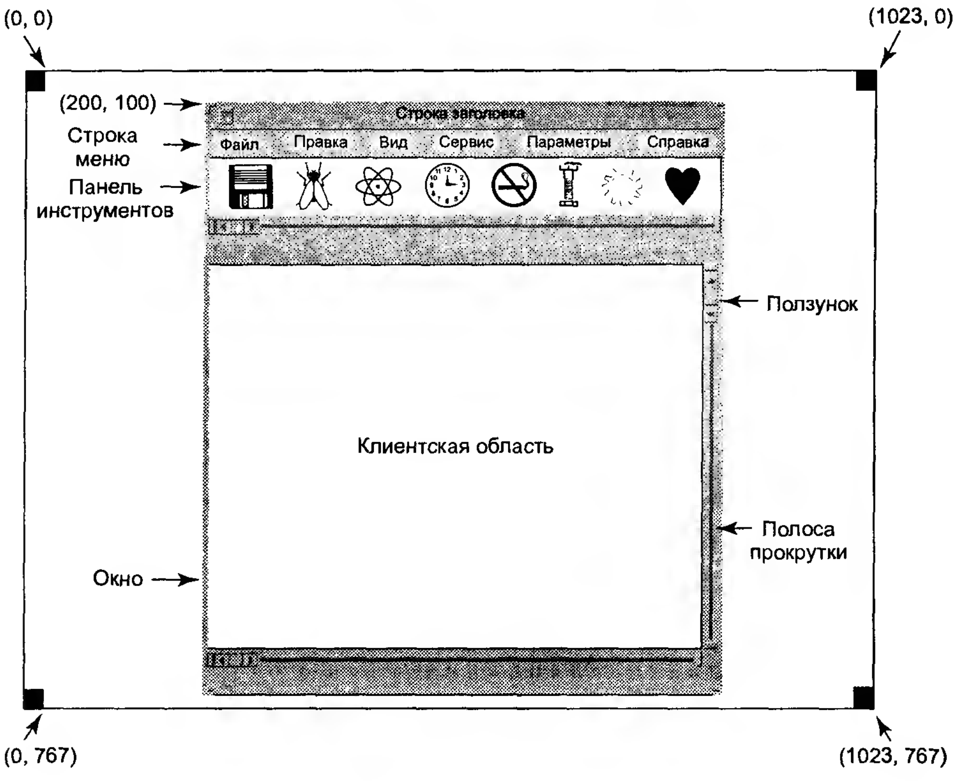


Рис. 5.34. Пример окна, расположенного на экране XGA-дисплея

При создании окна параметрами указывается, может ли это окно перемещаться пользователем, может ли пользователь изменять его размеры, будут ли у него полосы прокрутки и т. п. Главное окно большинства программ обычно можно перемещать, изменять его размеры и прокручивать его содержимое при помощи полос прокрутки с ползунками. Все эти возможности оказывают огромное влияние на способ написания программ. В частности, программы должны получать информацию об изменении размеров их окон и должны быть готовы в любое время перерисовать содержимое своих окон, даже когда они совсем этого не ждут.

Это привело к тому, что программы в системе Windows управляются сообщениями. Действия пользователя, включая работу с клавиатурой или мышью, перехватываются системой Windows и преобразуются в сообщения, адресуемые программе, владеющей окнами, к которым обращены действия пользователя. У каждой программы есть очередь сообщений, в которую направляются все сообщения, относящиеся к ее окнам. Главный цикл программы состоит из получения следующего

сообщения и обработки его с помощью вызова внутренней процедуры, соответствующей данному типу сообщений. В некоторых случаях система Windows может вызывать эти процедуры напрямую, минуя очередь сообщений. Эта модель принципиально отличается от используемой в системе UNIX модели процедурных программ, для взаимодействия с операционной системой обращающихся к системным вызовам.

Поясним программную модель, применяемую в системе Windows, на примере программы, приведенной в листинге 5.2. Здесь мы видим скелет основной программы для системы Windows. Она не закончена и не содержит обработки ошибок, но для наших целей она включает достаточно подробностей. Программа начинается с оператора включения файла заголовка *windows.h*, содержащего большое количество макросов, определений типов данных, констант, прототипов функций и другой информации, необходимой для программ, работающих в системе Windows.

Листинг 5.2. Скелет основной программы для системы Windows

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE h, HINSTANCE hprev, char *szCmd, int iCmdShow)
{
       WNDCLASS wndclass; /* обьект класса для этого окна */
       MSG msg; /* здесь сохраняются входящие сообщения */
       HWND hwnd; /* дескриптор (указатель) объекта окна */
       /* Инициализация обьекта wndclass */
       wndclass.lpfnWndProc = WndProc; /* адрес процедуры обратного вызова */
       wndclass.lpszClassName = "Program name"; /* текст строки заголовка */
       wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* загрузить пиктограмму программы
*/
       wndclass.hCursor = LoadCursor(NULL, IDC_ARROW); /* загрузить курсор мыши */
       RegisterClass(&wndclass); /* сообщить системе Windows об обьекте wndclass */
       hwnd = CreateWindow ( ... ) /* запросить память для окна */
       ShowWindow(hwnd, iCmdShow); /* отобразить окно на экране */
       UpdateWindow(hwnd); /* сообщение окну с требованием перерисовки */
       while (GetMessage(&msg, NULL, 0, 0)) { /* получить сообщение из очереди */
          TranslateMessage(&msg); /* транслировать сообщение */
          DispatchMessage(&msg);  /* послать сообщение msg соответствующей процедуре */
       }
       return(msg.wParam);
}
long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
       /* Здесь помещаются определения */
       switch (message) {
          case WM_CREATE: ... ; return ... ;  /* создать окно */
           case WM_PAINT: ... ; return ... ;  /* перерисовать содержимое окна */
          case WM_DESTROY: ... ; return ... ; /* уничтожить окно */
       }
       return(DefWindowProc(hwnd, message, wParam, lParam));/* по умолчанию */
}
```

Основная программа начинается с описания, содержащего ее имя и параметры. Макрос *WINAPI* представляет собой указание компилятору использовать определенное соглашение о передаче параметров и не будет нас дальше интересовать. Первый параметр, *h*, является дескриптором (описателем) экземпляра, используе-

мым для идентификации программы в системе. В определенном смысле интерфейс Win32 является объектно-ориентированным, это означает, что на этом уровне система Windows содержит объекты (например, программы, файлы и окна) с их состоянием и связанными с ними программами, называемыми **методами**, работающими с этим состоянием. Обращения к объектам производятся при помощи дескрипторов, в качестве которых используются указатели на объекты (адреса объектов). В данном случае *h* однозначно идентифицирует программу. Второй параметр более не используется и присутствует только ради обратной совместимости. Третий параметр, *szCmd*, представляет собой заканчивающийся нулевым байтом текст, содержащий командную строку запуска данной программы, даже если программа не была запущена из командной строки. Четвертый параметр, *iCmdShow*, сообщает, должно ли окно программы при запуске занять весь экран, часть экрана или окно должно быть минимизировано.

Это описание процедуры иллюстрирует широко используемое соглашение фирмы Microsoft, называемое **венгерской нотацией**. Название передразнивает так называемую польскую нотацию, придуманную польским логиком Й. Лукасевичем для представления алгебраических формул без использования скобок. Венгерская нотация была изобретена венгерским программистом из корпорации Microsoft Чарльзом Шимоньи. Она состоит в использовании первых нескольких символов идентификаторов для указания его типа. Среди допустимых символов и типов с (character — символ), w (word — слово, сегодня означает 16-разрядное целое без знака), i (integer — 32-разрядное целое со знаком), l (long — также 32-разрядное целое со знаком), s (string — строка), sz (строка, завершающаяся нулевым байтом), p (pointer — указатель), fn (function — функция) и h (handle — дескриптор). Так, *szCmd* представляет собой строку, завершающуюся нулем, а *iCmdShow* — целое число. Многие программисты полагают, что в подобном указании типа переменных в их именах мало пользы и оно лишь затрудняет чтение программ для системы Windows. В системе UNIX подобные соглашения не используются.

С каждым окном должен быть связан объект класса, описывающий его свойства. В листинге 5.2 таким объектом класса является *wndclass*. У объекта типа *WNDCLASS* 10 полей, четыре из которых инициализируются в листинге 5.2. В реальной программе остальные шесть также следует проинициализировать. Наиболее важным полем является *lpfnWndProc*, представляющим собой указатель типа long (то есть 32-разрядный) на функцию, обрабатывающую сообщения, направляемые окну. Другие поля, инициализируемые в этом примере, сообщают, какую пиктограмму использовать в окне, какой использовать курсор мыши, и задают строку заголовка.

После инициализации объекта *wndclass* вызывается процедура *RegisterClass*, чтобы передать его системе. Так, после этого вызова система Windows знает, какую процедуру вызывать для обработки различных событий, не устанавливаемых в очередь сообщений. Следующий вызов *CreateWindow* запрашивает у системы область памяти под структуру данных окна и возвращает дескриптор окна для последующих обращений к нему. Затем программа вызывает еще две системные процедуры, чтобы вывести окно на экран и нарисовать его содержимое.

Затем программа входит в главный цикл, состоящий из получения сообщения, выполнения с ним определенных преобразований и передачи его снова системе

на обработку. Для обработки сообщения система вызывает процедуру *WndProc*. В принципе все эти действия можно было организовать проще, но такая архитектура программ сложилась исторически, и теперь мы вынуждены с ней работать.

Следом за основной программой располагается процедура *WndProc*, обрабатывающая различные сообщения, посылаемые окну. Ключевое слово *CALLBACK* (обратный вызов) здесь, как и слово *WINAPI* выше, указывает используемое соглашение о передаче параметров процедуры. Первый параметр процедуры — дескриптор окна. Второй параметр содержит тип сообщения. В третьем и четвертом параметрах передается дополнительная информация.

Сообщения *WM_CREATE* и *WM_DESTROY*, извещающие о создании и уничтожении окна, посылаются в начале и в конце работы программы. Они предоставляют программе возможность, например, запросить буферы памяти для работы, а затем вернуть память системе.

Сообщение третьего типа, *WM_PAINT*, является указанием программе заполнить окно. Оно может посылаться приложению не только при первой прорисовке окна, но также и во время работы программы. В отличие от безоконных текстовых систем, программы в системе Windows не могут рассчитывать на то, что все, что они когда-либо выведут на экран, будет оставаться там вечно. Поверх одного окна может быть выведено или перетащено другое окно, открыты меню, диалоговые окна и всплывающие подсказки. При удалении этих элементов окно должно быть перерисовано. Чтобы сообщить программе, что окно следует перерисовать, система Windows посылает ей сообщение *WM_PAINT*. В нем также предоставляется информация о том, какая часть окна должна быть перерисована, что облегчает работу программе, позволяя перерисовать только часть окна.

Система Windows может заставить программу что-нибудь сделать двумя способами. Во-первых, система может послать программе сообщение, добавив его к очереди сообщений. Этот метод используется для ввода с клавиатуры, мыши и для сигналов от таймеров. Другой способ состоит в непосредственном вызове системой процедуры *WndProc*. Этот метод используется для всех прочих событий. Поскольку после полной обработки сообщения система Windows уведомляется об этом, она может воздержаться от отправки следующего сообщения до того, как будет обработано предыдущее. Таким образом удается избежать возникновения ситуации состязаний.

В системе Windows используется огромное количество сообщений различных типов. Чтобы избежать некорректного поведения программы в случае прихода неожиданного сообщения, в конце процедуры *WndProc* следует помещать обращение к системной процедуре *DefWindowProc*, таким образом, позволяя обработчику по умолчанию позаботиться обо всех остальных случаях.

Подведем итоги вышесказанного. Программа, работающая в системе Windows, обычно создает одно или несколько окон, для каждого из которых создается объект класса. С каждой программой связаны очередь сообщений и набор процедур обработки. В конечном итоге поведением программ управляют поступающие события, обрабатывающиеся специальными процедурами. Эта модель принципиально отличается от подхода, принятого в системе UNIX.

Собственно выводом на экран занимается пакет, состоящий из нескольких сот процедур, образующих вместе **интерфейс графических устройств** (**GDI**, Graphic Device Interface). Этот пакет может обрабатывать текст и все виды графики. Он разрабатывался с расчетом на независимость от платформ и устройств. Прежде чем программа может начать вывод в окне, она должна получить **контекст устройства**, представляющий собой внутреннюю структуру данных, содержащую свойства окна: текущий шрифт, цвет текста, цвет фона и т. д. Большинство процедур интерфейса GDI используют контекст устройства либо для вывода, либо для получения или установки свойств.

Существуют различные способы получения контекста устройства. Простой пример его получения и использования выглядит так:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

Первый оператор получает дескриптор контекста устройства, *hdc*. Во второй строке программы контекст устройства используется для вывода на экран строки текста. В параметрах процедуры указываются координаты начала печати (x, y), указатель на строку и ее длина. Третий вызов освобождает контекст устройства, сообщая системе, что программа закончила вывод. Обратите внимание, что контекст устройства *hdc* используется аналогично дескриптору файла в UNIX. Кроме того, следует заметить, что процедура *ReleaseDC* содержит избыточные параметры. Дескриптор контекста устройства *hdc* однозначно указывает окно. Использование избыточной информации, не требующейся для работы программы, довольно распространено в системе Windows.

Также следует заметить, что при получении контекста устройства *hdc* программа может писать только в клиентскую область окна, но не в заголовок строку состояния и т. п. В структуре данных контекста устройства внутренне поддерживается область отсечения. Любой вывод за пределы области отсечения игнорируется. Однако есть другая системная процедура, *GetWindowDC*, также позволяющая получить контекст устройства. Эта процедура устанавливает область отсечения, равную всему окну. Другие вызовы ограничивают область отсечения по-другому. Наличие в системе нескольких вызовов, выполняющих практически одно и то же, является еще одной характеристикой системы Windows.

Конечно, невозможно представить в этой книге полное описание работы с интерфейсом графических устройств. Читатели, которых интересует данная тема, могут найти дополнительную информацию в ссылках на литературу, приведенных выше. Тем не менее следует, возможно, сказать еще несколько слов о важности интерфейса GDI. Интерфейс GDI содержит различные процедуры, позволяющие получать и освобождать контексты устройств, получать информацию о контекстах устройств, получать и задавать атрибуты контекстов устройств (например, цвет фона), управлять такими объектами интерфейса GDI, как перья, кисти и шрифты, у каждого из которых есть свои атрибуты. Естественно, что интерфейс GDI содержит большое число процедур для собственно рисования на экране.

Процедуры графического вывода можно разделить на четыре категории: рисование прямых и кривых линий, вывод заполненных областей, управление растро-

выми изображениями и вывод текста. Пример вывода текста уже был приведен нами выше, поэтому давайте познакомимся с другими функциями. Вызов

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

рисует на экране заполненный прямоугольник, заданный координатами противоположных углов: (*xleft*, *ytop*) и (*xright*, *ybottom*). Например,

```
Rectangle(hdc, 2, 1, 6, 4);
```

выведет на экран прямоугольник, показанный на рис. 5.35. Толщина линии и цвет заливки задаются контекстом устройства. Другие обращения к интерфейсу GDI выглядят аналогично.

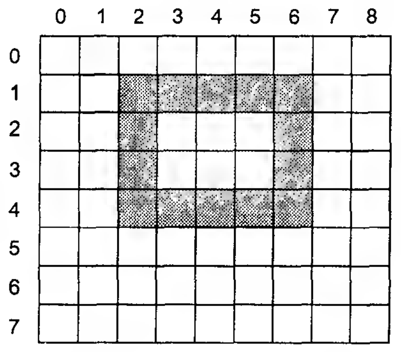


Рис. 5.35. Пример прямоугольника, нарисованного с помощью процедуры Rectangle. Каждый квадрат соответствует одному пикселу

Растровые изображения

Процедуры интерфейса GDI являются примерами векторной графики. Они используются для помещения на экран геометрических фигур и текста. Выводимые объекты легко могут быть масштабированы для вывода на большие или меньшие экраны, при условии что число пикселов на экране одинаково[1]. Вывод объектов также в большой степени независим от устройств. Набор обращений к процедурам GDI может быть собран в файл, описывающий сложные операции рисования. Такой файл в системе Windows называется **метафайлом**. Метафайлы широко применяются для передачи изображений от одной программы Windows к другой. Расширение у таких файлов *.wmf*.

Многие программы системы Windows позволяют пользователю скопировать изображение (или часть его) и поместить в буфер обмена Windows. Затем пользователь может перейти к другой программе и вставить содержимое буфера обмена в другой документ. Один из способов реализации данных действий состоит в представлении изображения в виде метафайла и помещении его в буфер обмена в формате *.wmf*. Существуют также и другие методы.

[1] Неясно, чем же тогда отличаются эти экраны. Интерфейс графических устройств Win32 GDI позволяет выводить все объекты не в пикселах, а в виртуальных единицах, что делает масштабирование действительно очень простым и эффективным. — *Примеч. перев.*

Не все изображения, с которыми работают компьютеры, могут быть созданы при помощи векторной графики. Например, в фотографиях и видеофильмах векторная графика не используется. Напротив, изображения в данном случае сканируются, в результате чего получается прямоугольная матрица цветных точек. Затем у каждой ячейки квадратной сетки измеряются и преобразуются в число значения интенсивности красного, зеленого и синего цвета. Все эти данные сохраняются в виде значения одного пиксела. Изображение, состоящее из таких пикселов, называют **растровым**. В системе Windows имеется широкий набор средств для работы с растровыми изображениями.

Растровые изображения применяются также для вывода текста. Одни из способов представления символов каким-либо шрифтом состоит в использовании небольших растровых изображений. В этом случае вывод текста на экран превращается в перемещение растровых изображений.

Для работы с растровыми изображениями часто используется процедура *bitblt*. Она вызывается следующим образом:

```
bitblt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

В простейшем случае она копирует растровое изображение из одного прямоугольника в другой (возможно, в другом окне). Первые три параметра указывают окно, в которое будет скопирован прямоугольник, и координаты в этом окне. Следом указываются ширина и высота прямоугольника, после которых задаются окно, из которого копируется прямоугольник, и координаты в нем. Обратите внимание, что у каждого окна есть своя система координат с началом (0,0) в верхнем левом углу окна. Последний параметр будет описан ниже. Эффект вызова

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

показан на рис. 5.36. Обратите внимание, что скопированной оказалась вся область 5×7 пикселов символа А, включая цвет фона.

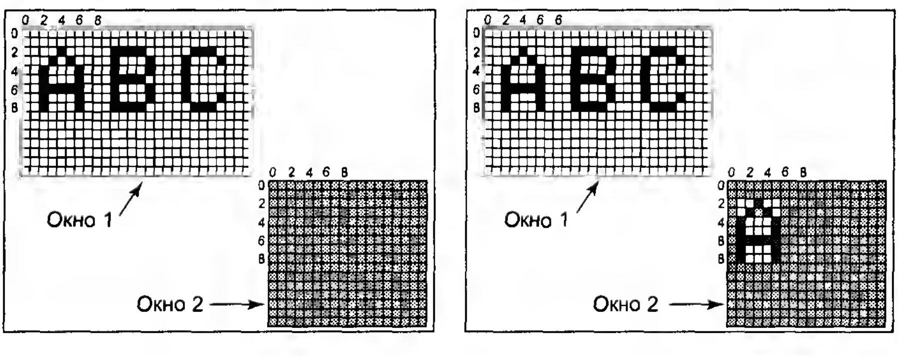


Рис. 5.36. Копирование растровых изображений с помощью процедуры BitBlt: до (*а*); после (*б*)

Процедура *BitBlt* может не только копировать растровые изображения. Последний параметр процедуры дает возможность выполнять с растровыми изображениями побитовые логические операции, что позволяет объединять растровые

изображения. Например, к ним можно применить логическую операцию ИЛИ или сложение по модулю 2.

Недостатком растровых изображений является сложность их масштабирования. Символ размера 8×12 точек выглядит вполне разумно на экране с разрешением 640×480 пикселов. Однако при копировании этого растрового изображения на печатаемый лист с разрешением 1200 точек на дюйм, что соответствует разрешению 10200×13200 точек на листе, ширина символа (8 пикселов) окажется равной 8/1200 дюйма или около 0,17 мм. Кроме того, копирование с одного устройства на другое усложняется различными цветовыми свойствами устройств.

По этой причине системой Windows также поддерживается структура данных, называемая **аппаратно-независимым растровым изображением** (**DIB**, Device-Independent Bitmap). Изображение такого формата хранится в файлах с расширением *.bmp.* В этих файлах помимо пикселов хранятся информационные заголовки и цветовая таблица. Такие данные облегчают копирование растровых изображений между несхожими устройствами.

Шрифты

В системе Windows, предшествовавших версии Windows 3.1, символы представлялись в виде растровых изображений и копировались на экран или принтер с помощью процедуры *BitBlt.* Проблема была в том, что, как уже было продемонстрировано, растровое изображение, годящееся для экрана, слишком мало для принтера. К тому же для каждого символа каждого размера требовался отдельный растр. Другими словами, при наличии растрового изображения символа А размером в 10 точек нет способа вычислить его для 12-точечного шрифта. Для каждого символа каждого шрифта размером от 4 до 120 точек понадобится огромное количество растровых изображений. Вся система при этом оказывалась крайне неуклюжей.

Решение этой проблемы было предложено при помощи шрифтов TrueType, представляющих собой не растровые изображения, а контуры символов. Каждый символ определяется последовательностью точек по своему периметру. С помощью такой системы символы легко увеличиваются и уменьшаются. Все, что нужно для этого сделать — это умножить координаты каждой точки на один и тот же множитель. Это позволяет масштабировать символы, задавая любой, даже нецелый, размер шрифта. После задания нужного размера точки периметра символа соединяются друг с другом при помощи нехитрого алгоритма, напоминающего картинку-загадку для детей (в последнее время для получения более гладких результатов используются сплайны). Когда контур полностью обведен, символ может быть закрашен. Пример нескольких символов трех различных размеров показан на рис. 5.37.

Таким образом, промасштабировав контур символа и преобразовав его в растровое изображение, можно гарантировать, что символы, изображаемые на экране и печатаемые на принтере, будут близки, насколько это возможно, отличаясь только ошибками квантования. Чтобы еще более увеличить качество, можно каждый символ снабдить подсказками, помогающими в выполнении растеризации. Например, обе засечки у буквы Т должны быть идентичными, что может не получиться вследствие ошибок округления.



Рис. 5.37. Несколько примеров контуров символов различных размеров

Сетевые терминалы

Сетевые терминалы используются для того, чтобы соединить удаленного пользователя с компьютером по локальной или глобальной сети. Существует две различные философские концепции, касающиеся способа работы сетевых терминалов. Одна точка зрения заключается в том, что сетевой терминал должен обладать огромной вычислительной мощностью и памятью, что должно позволить работать на нем сложным протоколам и снизить объем данных, пересылаемых по сети. (**Протоколом** называется набор запросов и ответов, о которых договариваются отправитель и получатель, чтобы общаться по сети или другому интерфейсу.) Другая точка зрения состоит в том, что терминал должен быть максимально простым и дешевым, в основном занимающимся лишь отображением пикселов на экране. В следующих двух разделах мы на примерах обсудим каждую точку зрения. Сначала мы познакомимся с изощренной системой X Windows, затем рассмотрим минимальный терминал SLIM.

Система X Window

Крайняя степень интеллектуального терминала представляет собой терминал, содержащий центральный процессор, такой же мощный, как и у основного компьютера, с мегабайтами памяти, клавиатурой и мышью. Терминалом такого типа является **X-терминал**, на котором работает система **X Window System** (часто называемая просто X), разработанная в Массачусетсском технологическом институте как часть проекта Athena. X-терминал представляет собой компьютер, на котором работают X-программы и который взаимодействует с программами, работающими на удаленном компьютере.

Программа, работающая на X-терминале, собирающая ввод с клавиатуры или мыши и принимающая команды от удаленного компьютера, называется **X-сервером**. Она должна следить за тем, которое из окон выбрано в данный момент (то, над которым находится курсор мыши). Таким образом X-сервер узнает, которому клиенту направлять ввод с клавиатуры. X-сервер общается по сети с **X-клиентами**, работающими на удаленном хосте. Он посылает им ввод с клавиатуры и мыши, а также принимает от них команды отображения.

Может показаться странным наличие X-сервера на терминале и клиентов на удаленном хосте, но работа X-сервера состоит в отображении битов, поэтому он должен находиться близко к пользователю. С точки зрения программы, клиент велит серверу выполнить те или иные действия, например вывести текст или отобразить геометрическую фигуру. Сервер (в терминале), как и все серверы, просто делает то, что ему велят. Схема клиента и сервера показана на рис. 5.38.

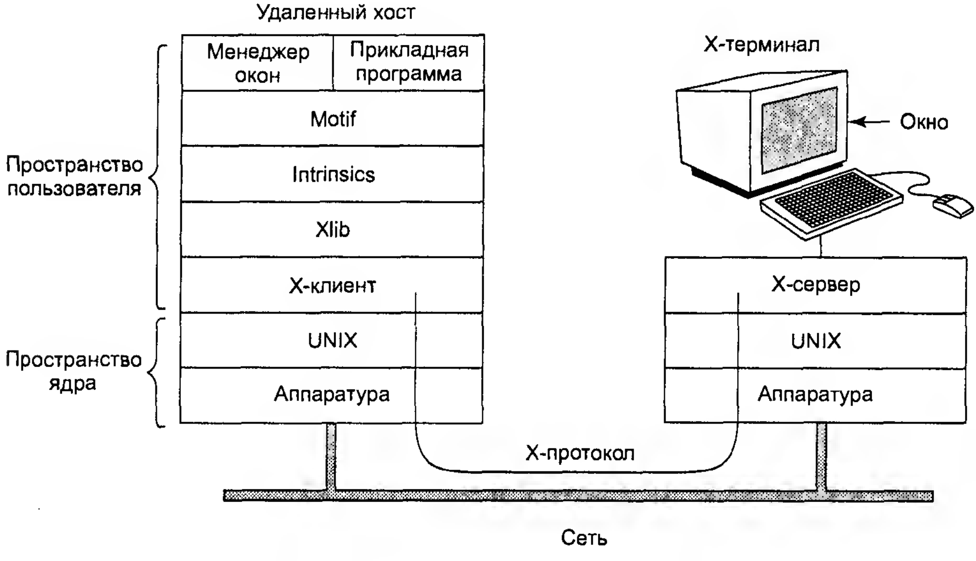


Рис. 5.38. Клиенты и серверы в системе X Windows

Система X Windows может работать поверх UNIX или другой операционной системы. Действительно, на многих разновидностях системы UNIX в качестве стандартной оконной системы работает X Windows, даже на автономных машинах или для доступа к удаленным машинам через Интернет. Что система X Windows определяет в действительности — это протокол между X-клиентом и X-сервером, как показано на рис. 5.38. Не имеет значения, работают ли клиент и сервер на одной машине, соединены ли они локальной сетью на расстоянии сотни метров или между ними тысячи километров, и они обмениваются информацией по Интернету. Протокол и операционная система идентичны во всех случаях.

Система X Windows представляет собой всего лишь оконную систему. Она не является полным графическим интерфейсом пользователя. Для предоставления пользователю полного графического интерфейса поверх системы X Windows запускаются другие уровни программного обеспечения. Одним таким уровнем являет-

ся **Xlib**, представляющий собой набор библиотечных процедур, необходимых для предоставления доступа к функциям системы X Windows. Эти процедуры образуют базис системы X Windows, и мы их изучим ниже, но они слишком примитивны для большинства программ пользователя, чтобы их применять напрямую. Например, они сообщают о каждом щелчке мыши отдельно, поэтому, чтобы определить, являются ли два последовательных щелчка двойным щелчком, требуется дополнительная обработка этих сообщений на уровне выше Xlib.

Чтобы облегчить программирование в системе X Windows, вместе с ней поставляется набор инструментальных средств, называющийся **Intrinsics** (встроенные средства). Этот уровень управляет кнопками, полосами прокрутки и другими элементами графического интерфейса пользователя, называемыми **widget** («штуковина»). Для создания настоящего графического интерфейса пользователя, однородно воспринимаемого пользователем, требуется еще один уровень. Наиболее популярный стандарт графического интерфейса пользователя называется **Motif**. Большинство приложений пользуются именно функциями интерфейса Motif, а не Xlib.

Также следует заметить, что управление окнами не является частью самой системы X Windows. Из системы оно вынесено преднамеренно. Вместо нее созданием, удалением и перемещением окон по экрану управляет отдельный X-клиентский процесс, называемый **оконным менеджером**. Для этого он посылает команды X-серверу. Часто он работает на той же машине, что и X-клиент, но теоретически он может работать где угодно.

Такой модульный дизайн, состоящий из нескольких уровней и большого количества программ, делает систему X Windows в высочайшей степени переносимой и гибкой. Она была установлена на большинство версий системы UNIX, включая Sun Solaris, BSD, AIX, Linux и т. п., что предоставило разработчикам стандартный интерфейс пользователя на различных платформах. Система X Windows была также установлена на другие операционные системы. Напротив, в системе Windows управление окнами и графический интерфейс пользователя смешаны вместе в интерфейсе GDI и располагаются в ядре, в результате чего управление ими усложняется. Например, графический интерфейс пользователя системы Windows 98 по-прежнему в основном остается 16-разрядным, спустя более чем десять лет после появления 32-разрядных процессоров Intel.

Взглянем теперь на систему X Windows с точки зрения уровня Xlib. Когда в системе X Windows запускается программа, она устанавливает соединение с одним или более X-серверами — мы будем называть их рабочими станциями, даже если они располагаются на той же машине, что и сама программа. Система X Windows считает это соединение надежным, в том смысле, что потерянные сообщения и дубликаты сообщений обрабатываются сетевым программным обеспечением и программе не нужно беспокоиться об ошибках связи. Обычно для соединения клиента и сервера используется пара протоколов TCP/IP.

По соединению передаются следующие четыре типа сообщений:

1. Команды рисования от программы к рабочей станции.
2. Ответы рабочей станции на запросы программы.
3. Объявления о различных событиях, таких как ввод с клавиатуры или мыши и т. п.
4. Сообщения об ошибках.

Большинство команд рисования посылаются от программы к рабочей станции как сообщения, на которые не ожидается никакого ответа. Причина этого в том, что при нахождении клиента и сервера на различных машинах для прохождения сообщения и ответа на него по сети может потребоваться значительное время. Блокирование прикладной программы в течение этого периода времени лишь сильно замедлило бы ее выполнение без особой на то необходимости. С другой стороны, когда программе требуется информация от рабочей станции, она просто должна подождать, пока не придет ответ.

Как и Windows, система X Windows в значительной степени управляется событиями. Поток событий направляется от рабочей станции к программе, обычно в ответ на некое действие пользователя, например ввод с клавиатуры, перемещение мыши или открытие окна. Каждое сообщение имеет 32 байт в длину, из которых первый байт содержит тип сообщения, а остальные 31 байт содержат дополнительную информацию. Существует несколько десятков сообщений, но программе посылаются только те сообщения, о которых она заявила, что хочет сама их обрабатывать. Например, если программу не интересуют такие события, как отпускание клавиш, то о таких событиях ей не сообщается. Как и в Windows, события устанавливаются в очередь, из которой их читает программа. Однако в отличие от Windows операционная система никогда не вызывает процедуры прикладной программы сама. Операционная система даже не знает, какая процедура, какие события обрабатывает.

Ключевой концепцией системы X Windows является **ресурс**. Ресурсом называется структура данных, хранящая определенную информацию. Прикладные программы создают ресурсы на рабочих станциях. Ресурсы могут использоваться совместно несколькими процессами на рабочей станции. Обычно ресурсы живут недолго и не переживают перезагрузки рабочей станции. К типичным ресурсам относятся окна, шрифты, карты цветов (цветовые палитры) карты пикселов (растровые изображения), курсоры и графические контексты. Последние используются для связи свойств с окнами и концептуально схожи с контекстами устройств в Windows.

Грубый незаконченный скелет программы для системы X Windows приведен в листинге 5.3. В начале этой программы включаются необходимые файлы заголовков, после чего объявляются переменные. Затем она устанавливает соединение с X-сервером, указанным как параметр процедуры *XOpenDisplay*. После этого программа запрашивает память для ресурса окна и сохраняет дескриптор окна в переменной *win*. В действительности здесь должна производиться определенная инициализация. Затем программа сообщает оконному менеджеру о существовании нового окна.

Листинг 5.3. Скелет прикладной программы для системы X Windows

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
main(int argc, char * argv[])
{
        Display disp;                          /* идентификатор сервера */
        Window win;                            /* идентификатор окна */
        GC gc;                                 /* идентификатор графического контекста */
        XEvent event;                          /* место хранения для одного события */
```

```
    int running = 1;
    disp = XOpenDisplay("display_name");   /* установить соединение с X-сервером */
    win = XCreateSimpleWindow(disp, ... ); /* запросить память для нового окна */
    XSetStandardProperties(disp, ...);     /* объявить об окне оконному менеджеру */
    gc = XCreateGC(disp, win, 0, 0);       /* создать графический контекст */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                 /* отобразить окно; послать событие Expose */
    while (running) {
        XNextEvent(disp, &event);          /* получить следующее событие */
        switch (event.type) {
            case Expose: ...; break;       /* перерисовать окно */
            case ButtonPress: ...; break;  /* обработать щелчок мыши */
            case Keypress: ...; break;     /* обработать ввод с клавиатуры */
        }
    }
    XFreeGC(disp, gc);                     /* освободить графический контекст */
    XDestroyWindow(disp, win);             /* вернуть память, занимаемую окном */
    XCloseDisplay(disp);                   /* разорвать сетевое соединение */
}
```

Вызов *XCreateGC* создает графический контекст, в котором сохраняются свойства окна. В более полной программе они, возможно, в этом месте будут проинициализированы. Следующая строка, обращаясь к системной процедуре *XSelectInput*, сообщает X-серверу, какие события программа собирается обрабатывать сама. В данном случае ее интересуют щелчки мыши, нажатия на клавиши и открытие окон. В действительности программы обычно обрабатывают также и другие события. Наконец, вызов *XMapRaised* отображает новое окно на экран поверх остальных окон. С этого момента окно становится видимым на экране.

Главный цикл состоит из двух операторов и логически значительно проще, чем соответствующий цикл в системе Windows. Первый оператор здесь получает событие, а второй осуществляет диспетчеризацию событий, направляя их на обработку в соответствии с типами. Когда событие сообщает программе, что ее выполнение завершается, значение логической переменной *running* устанавливается равным 0 и выполнение цикла прекращается. Перед тем как закончить свою работу, программа освобождает графический контекст, окно и соединение.

Следует отметить, что далеко не всем программистам нравится графический интерфейс пользователя. Многие предпочитают традиционный интерфейс командной строки, вроде обсуждавшегося в разделе «Программное обеспечение ввода» данной главы. Системой X Windows интерфейс командной строки поддерживается при помощи программы *xterm*. Эта программа эмулирует старый «умный» терминал VT102, поддерживающий полный набор ESC-последовательностей. В этих окнах без каких бы то ни было переделок работают текстовые редакторы *vi* и *emacs*, а также другое программное обеспечение, использующее базу данных termcap.

Сетевой терминал SLIM

В течение многих лет основная компьютерная парадигма колебалась между централизованными и децентрализованными вычислениями. Первые компьютеры, такие как ENIAC, были на самом деле персональными компьютерами, хотя и очень большими, поскольку только один пользователь мог работать на таком компьюте-

ре в каждый момент времени. Затем появились системы разделения времени, в которых много удаленных пользователей одновременно работали на большом центральном компьютере. Потом наступила эра персональных компьютеров, то есть у пользователей снова появились собственные компьютеры.

Хотя децентрализованная модель персональных компьютеров обладает определенными преимуществами, у нее есть также серьезные недостатки, которые только последнее время начинают серьезно рассматриваться. Возможно, самая большая проблема персональных компьютеров состоит в том, что у каждого ПК имеется большой жесткий диск и сложное программное обеспечение, которым требуется управлять. Например, при выходе новой версии операционной системы на каждой машине отдельно потребуется выполнить обновление программного обеспечения, что может занять довольно много времени. В большинстве корпораций затраты на выполнение подобного рода работ сопоставимы со стоимостью оборудования и самого программного обеспечения. Домашним пользователям компьютеров не нужно платить самим себе за эту работу, однако далеко не все пользователи могут выполнить ее корректно, и еще меньшему числу пользователей нравится этим заниматься. В централизованных системах программное обеспечение должно быть обновлено лишь на одной машине или небольшом количестве машин, для обслуживания которых у корпорации обычно имеется штат экспертов, способных выполнить эту работу.

Помимо периодического обновления программного обеспечения, пользователям также следует регулярно архивировать свои гигабайтные файловые системы, хотя мало кто этим занимается. Когда случается несчастье, пользователям остается только причитать. В централизованной системе резервные копии могут каждую ночь записываться автоматами на магнитные ленты.

Еще одно преимущество централизованной системы состоит в том, что в этом случае упрощается совместное использование ресурсов. В системе из 64 удаленных пользователей, у каждого из которых есть по 64 Мбайт оперативной памяти, большую часть времени значительная часть памяти будет оставаться неиспользуемой. В централизованной системе с 4 Гбайт ОЗУ никогда не случается так, что какому-либо пользователю временно требуется много оперативной памяти, но он не может ее получить, так как она кем-то занята. Тот же аргумент справедлив для дискового пространства и других ресурсов.

Вероятно, мы не сильно ошибемся, если скажем, что большинству пользователей нужна высокопроизводительная интерактивная компьютерная среда, но они не хотели бы заниматься администрированием компьютера. Это заключение заставило многих исследователей вспомнить о системах разделения времени с «глупыми» терминалами (теперь вежливо называемыми **«тонкими» клиентами**), вполне соответствующих современным представлениям о терминалах. Система X Windows была одним из шагов в этом направлении, но X-сервер все еще остается сложной системой, состоящей из нескольких мегабайтов программного обеспечения, которое требуется периодически обновлять. Идеалом была бы высокопроизводительная интерактивная компьютерная система, в которой машина пользователя вообще не имела бы программного обеспечения. И что интересно, такая цель достижима. Ниже мы опишем одну такую систему, разработанную исследователями корпорации Sun

Microsystems и Стэнфордского университета. Сегодня эта система распространяется на коммерческой основе корпорацией Sun [295].

Система получила название **SLIM** (Stateless Low-level Interface Machine — машина низкоуровневого интерфейса без состояний). В основе идеи лежит традиционная схема централизованного разделения времени, показанная на рис. 5.39. Клиентские машины представляют собой просто «глупые» растровые дисплеи с разрешением 1280×1024 точки, с клавиатурой и мышью, но без программного обеспечения, устанавливаемого пользователем. Все это в большой степени соответствует духу старых «интеллектуальных» алфавитно-цифровых терминалов, у которых не было никакого программного обеспечения, а только некоторое количество программно-аппаратных средств для интерпретации ESC-последовательностей. Терминалы такого типа, не обладающие мощными вычислительными способностями, называются **«тонкими» клиентами**.

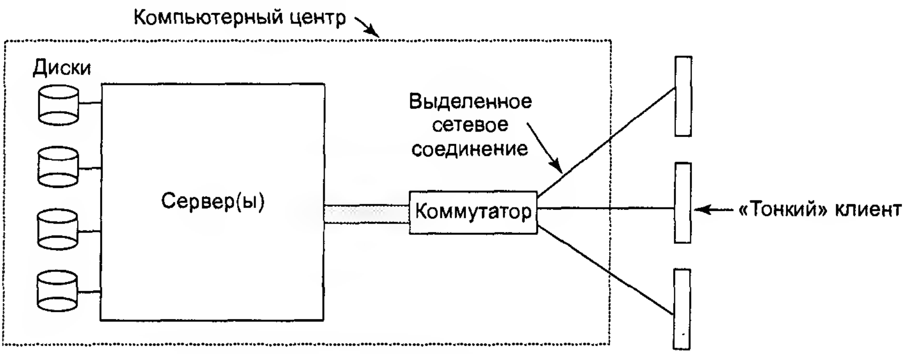


Рис. 5.39. Архитектура терминальной системы SLIM

Простейшая модель, состоящая в передаче сервером растровых изображений по сети «тонким» клиентам, не работает. Для этого потребуется пропускная способность для каждой выделенной линии около 2 Гбайт/с, что слишком много для современных сетей. Следующая по простоте модель, заключающаяся в хранении образа экрана в буфере кадра на терминале и обновлении экрана локально, является намного более обещающей. В частности, если центральный сервер будет хранить копию буфера кадра каждого терминала и посылать только обновления (изменения), требуемая пропускная способность будет уже не столь велика. Таким образом работают «тонкие» клиенты системы SLIM.

В отличие от X-протокола, состоящего из сотен сложных сообщений для управления окнами, рисования геометрических фигур и отображения текста различными шрифтами, у протокола SLIM есть всего пять сообщений от сервера к терминалу, перечисленные в табл. 5.6 (помимо них имеется еще небольшое количество не приведенных в таблице управляющих сообщений). Сообщение SET просто заменяет прямоугольник в буфере кадра новыми пикселами. Каждый заменяемый пиксел требует трех байтов в сообщении, чтобы указать его полное (24-разрядное) значение цвета. В принципе этого сообщения достаточно, чтобы выполнить всю работу. Остальные сообщения представляют собой всего лишь оптимизацию.

Таблица 5.6. Сообщения от сервера к терминалу протокола SLIM

Сообщение	Значение
SET	Обновить прямоугольник новыми пикселами
FILL	Заполнить прямоугольник пикселами одного значения
BITMAP	Растянуть растровое изображение, чтобы заполнить прямоугольник
COPY	Копировать прямоугольник из одной части буфера в другую
CSCS	Преобразовать прямоугольник в RGB из телевизионной системы цветов (YUV)

Сообщение FILL заполняет целый прямоугольник пикселами одного значения. Эта команда используется для заполнения однородных фоновых поверхностей. Сообщение BITMAP заполняет целый прямоугольник, повторяя одно растровое изображение, содержащееся в сообщении. Эта команда полезна для заполнения мозаичного фона.

Сообщение COPY велит терминалу скопировать прямоугольник из одной части экранного буфера в другую. Она наиболее всего полезна при скроллинге экрана и перемещении окон.

Наконец, сообщение CSCS преобразует цвета из системы YUV, используемой в США в телевизионной системе NTSC, в систему RGB, применяемую компьютерными мониторами. В первую очередь эта команда может использоваться при передаче необработанного видеокадра от сервера терминалу. Алгоритм преобразования несложен, но занимает много времени, поэтому эту работу лучше поручить терминалам. Если терминалы не будут использоваться для просмотра видеоформата NTSC, то это сообщение не понадобится.

В целом идея «глупых» «тонких» клиентов оказывается либо работоспособной, либо неработоспособной в зависимости от требующейся и доступной производительности сети и серверов, измерением которой интенсивно занимались Шмидт с коллегами [295]. В исследуемом ими прототипе как на участке между серверами и коммутатором, так и для соединений между коммутатором и терминалами использовалась 100-Мбитная коммутируемая быстрая сеть Ethernet. В принципе сервер с коммутатором можно было соединить и гигабитной сетью, так как весь этот участок находился в центральном вычислительном зале.

Первые измерения касались вывода эха символа на экран. Каждый введенный с клавиатуры символ посылается на сервер, который вычисляет координаты пикселов, требующих обновления, чтобы поместить на экран символ в нужную позицию, с правильными цветами и шрифтом. Измерения показали, что для появления символа на экране требуется 0,5 мс. Для сравнения: на локальной рабочей станции из-за буферизации в ядре это время составляет около 30 мс.

Остальные тесты измеряли производительность системы. При этом пользователи работали с современными интерактивными прикладными программами, такими как Adobe Photoshop (программа для ретуширования фотографий), Adobe Framemaker (настольная издательская система) и Netscape (web-браузер). Было замечено, что половина команд пользователей требовала обновления менее 10 000 пикселов, что в несжатом виде составляет 30 000 байт. На скорости 10 Мбит/с для передачи по кабелю 10 000 пикселов требуется 2,4 мс. Еще 2,7 мс нужно для помещения их в буфер кадра по прибытии, итого 5,1 мс (это время может немного

варьироваться в зависимости от обстоятельств). Поскольку время реакции человека составляет около 100 мс, обновления кажутся практически мгновенными. Даже изменения больших объемов данных воспринимались бы как почти мгновенные. Более того, при использовании сжатия для более чем 85 % обновлений экрана требуется передача менее 30 000 байт.

Эксперименты были повторены с 10-Мбитной сетью, 1-Мбитной сетью и 128-Кбитной сетью. При использовании 10-Мбитной сети система была практически мгновенной. На 1-Мбитной сети результаты оставались хорошими. 128-Кбитная сеть оказалась слишком медленной для таких задач. Поскольку соединения с пропускной способностью 1 Мбит/с становятся все более распространенными благодаря сетям кабельного телевидения и ADSL (Asymmetric Digital Subscriber Loop — асимметричная цифровая абонентская линия), то, похоже, что эта технология уже может применяться как для домашних пользователей, так и для деловых клиентов.

Управление режимом энергопотребления

Первый универсальный электронный компьютер ENIAC (Electronic Numerical Integrator and Calculator — электронный цифровой интегратор и калькулятор) состоял из 18 000 электронных ламп и потреблял 140 кВт. В результате счета за электричество были довольно высоки. После изобретения транзистора потребление электроэнергии снизилось на несколько порядков, в результате чего компьютерная промышленность потеряла интерес к экономии электроэнергии. Однако сегодня по различным причинам управление режимом электропитания снова оказалось в центре внимания, и занимается этим в большой степени операционная система.

Начнем с настольного персонального компьютера. Обычный настольный персональный компьютер содержит 200-ваттный блок питания (КПД такого блока питания составляет около 85 %. Это означает, что около 15 % энергии преобразуется в тепло уже в блоке питания). Если одновременно включить питание 100 млн таких машин по всему миру, то вместе они будут использовать около 20 ГВт. Эта мощность примерно соответствует мощности 20 атомных электростанций среднего размера. Если бы потребление электроэнергии компьютерами можно было уменьшить вдвое, то мы могли бы закрыть 10 атомных электростанций. С точки зрения охраны окружающей среды закрытие 10 атомных электростанций (или эквивалентного количества электростанций, работающих на сжигаемом топливе) является большим достижением, и за это стоит побороться.

Другая область, где потребление электроэнергии также является важным вопросом, — это переносные компьютеры, питаемые от батарей, включая, среди прочих, ноутбуки, лэптопы, палмтопы и web-блокноты. Основой проблемы является то, что батареи не могут удерживать количество энергии, достаточное для долгой работы компьютера. Обычное время, на которое хватает заряда аккумуляторов, не превышает нескольких часов. Несмотря на широкомасштабные исследования, проводимые различными компаниями, специализирующимися в области производства электрических элементов, прогресс в этой области практически нулевой. Для индустрии, привыкшей к удвоению производительности через каждые 18 месяцев

(закон Мура), полное отсутствие прогресса выглядит как нарушение законов физики. В результате создание компьютеров, потребляющих меньше энергии, чтобы существующих батарей хватало на более долгий срок, является задачей номер один для многих исследователей. Как мы увидим, операционная система здесь играет основную роль.

К снижению потребления электроэнергии существует два основных подхода. Первый из них заключается в выключении операционной системой тех частей компьютера (главным образом, устройств ввода-вывода), которые не используются в данный момент. Второй подход состоит в снижении потребления энергии прикладными программами, возможно, за счет снижения качества восприятия пользователем. Мы по очереди рассмотрим оба подхода, но сначала скажем несколько слов о технических средствах компьютера с точки зрения энергопотребления.

Аппаратный аспект

Электрические элементы подразделяются на два основных типа: одноразовые и перезаряжаемые (аккумуляторы). Одноразовые электрические элементы (например, AAA, AA и D) могут использоваться для питания небольших портативных устройств размером с ладонь, но у них недостаточно энергии, чтобы питать лэптопы с большими яркими экранами[1]. Аккумуляторные батареи, напротив, могут хранить достаточно энергии для питания лэптопов в течение нескольких часов. Обычно для этой цели использовались никель-кадмиевые элементы, но в последнее время кадмий в этих элементах был заменен металлическими гидридами, в результате чего время жизни аккумуляторов увеличилось, и, кроме того, новые элементы наносят меньший ущерб окружающей среде, когда их в конце концов приходится выбрасывать. Литиевые ионные батареи еще лучше, так как их можно перезаряжать, не дожидаясь полной разрядки, но их емкость также ограничена.

Основной метод, предпринимаемый многими производителями компьютеров для продления жизни батарей, состоит в проектировании центрального процессора, памяти и устройств ввода-вывода, способных находиться в нескольких режимах энергопотребления: включенном, режиме ожидания, режиме глубокой «зимней спячки», называемой также гибернацией, и в выключенном состоянии. Чтобы устройством можно было пользоваться, оно должно быть включено. Если устройство не используется в течение короткого интервала времени, оно может быть переведено в режим низкого энергопотребления. Если интервал времени, в течение которого устройство не используется, более долгий, потребление энергии устройством может быть снижено в еще большей степени. Обычно чем в более глубокую «спячку» переводится устройство, тем больше времени требуется, чтобы вывести его оттуда. Наконец, когда устройство выключено, оно не потребляет энергии вовсе. Не у всех устройств есть все эти состояния, но если устройство поддерживает режимы низкого энергопотребления, то именно операционная система должна управлять переводом устройства из одного режима в другой в соответствующий момент.

[1] Дело, скорее, в размерах элементов, а не в одноразовости. Обычно, наоборот, аккумуляторы могут хранить чуть меньше заряда, чем одноразовые элементы. В противном случае все «Энерджайзеры» и «Дюраселы» были бы перезаряжаемыми. Аккумуляторы просто удобнее и дешевле с учетом возможности их перезарядки. — *Примеч. перев.*

У некоторых компьютеров есть две или даже три кнопки питания. Одна из них может перевести весь компьютер в режим низкого энергопотребления, из которого он может быть быстро выведен простым нажатием любой клавиши или перемещением мыши. Другая кнопка может переводить компьютер в более глубокую «спячку», возвращение в активное состояние из которой может занять значительно больше времени. В обоих случаях эти кнопки обычно всего лишь посылают особые сигналы операционной системе, которая выполняет все необходимые действия. В некоторых странах закон требует, чтобы все электрические устройства в целях безопасности оснащались механическим выключателем электропитания. Для соответствия этому закону может понадобиться еще одна кнопка.

Управление режимом энергопотребления поднимает ряд вопросов, с которыми операционная система должна иметь дело. Среди этих вопросов такие: какие устройства могут управляться операционной системой? Есть ли у этих устройств два состояния, включенное и выключенное или имеются еще и промежуточные? Сколько мощности сберегается в различных режимах низкого энергопотребления? Расходуется ли энергия на перезапуск (разгон) устройства? Должен ли сохраняться контекст при переводе устройства в режим низкого энергопотребления? Сколько времени занимает активизация устройства? Ответы на все эти вопросы различны для разных устройств, поэтому операционная система должна уметь работать с широким спектром возможных вариантов и комбинаций.

Многие исследователи изучали лэптопы, пытаясь понять, на что тратится энергия. Ли с сотоварищами в 1994 году произвел измерения энергопотребления в различных режимах работы и пришел к выводу, проиллюстрированному в табл. 5.7 [206]. Лорх и Смит в 1998 году произвели замеры на других машинах и получили несколько отличные результаты, также показанные в табл. 5.7 [215]. Вайзер с коллегами в 1994 году тоже занимались сходными измерениями, но эта группа исследователей не опубликовала численных значений [357]. Они просто заявили, что основными потребителями электроэнергии в переносных компьютерах (в порядке убывания) являются экран, жесткий диск и центральный процессор. Хотя результаты, полученные разными исследователями, и не соответствуют друг другу в точности, возможно потому, что использовались различные марки компьютеров, тем не менее ясно, что именно экран, жесткий диск и центральный процессор являются теми устройствами, потребление энергии которыми следует пытаться снизить.

Таблица 5.7. Энергопотребление различных частей лэптопа

Устройство	Ли и его коллеги (1994)	Лорх и Смит (1998)
Дисплей	68 %	39 %
Центральный процессор	12 %	18 %
Жесткий диск	20 %	12 %
Модем		6 %
Звуковая плата		2 %
Память	0,5 %	1 %
Другое		22 %

Аспект операционной системы

Операционная система играет ключевую роль в управлении режимом энергопотребления. Она управляет всеми устройствами, поэтому ей приходится решать, какое устройство и когда выключать. Если выключаемое устройство потребуется снова в течение короткого интервала времени, выключения устройства будут лишь приводить к раздражающим задержкам при его включении. С другой стороны, если операционная система будет слишком долго ждать, прежде чем выключить устройство, это приведет к напрасной растрате электроэнергии.

Хитрость состоит в том, чтобы найти алгоритмы, позволяющие операционной системе принимать правильные решения, касающиеся энергопотребления. Сложность в том, что понятие «правильного» решения в большой степени субъективно. Для одного пользователя может оказаться вполне приемлемым, если после того, как он в течение 30 с не использует компьютер, потребуется 2 с, чтобы получить ответ на нажатую клавишу. Другой же пользователь, оказавшись в такой же ситуации, станет ругаться до посинения. Если у компьютера не будет устройства аудиоввода, он не сможет отличить одного пользователя от другого.

Дисплей

Рассмотрим теперь устройства, являющиеся самыми крупными потребителями электроэнергии, чтобы понять, что можно сделать с каждым из них. Больше всех устройств электроэнергии потребляет дисплей. Чтобы изображение было ярким и контрастным, экран должен иметь подсветку, а для этого требуется значительная энергия. Многие операционные системы пытаются сберегать энергию, отключая дисплей, когда в течение определенного интервала времени не наблюдается активности со стороны пользователя. Обычно пользователь может сам задать значение интервала времени, после которого следует выключать дисплей. Таким образом, пользователь может сам решить этот вопрос, выбирая между часто гаснущим дисплеем и быстро садящимися аккумуляторами. Из этого режима пониженного энергопотребления дисплей может быть выведен обратно в активное состояние практически мгновенно нажатием одной клавиши, при этом изображение регенерируется из видеопамяти.

Один вариант усовершенствования был предложен Флинном и Сатьянарайянаном [117]. Они предложили разделить экран на несколько отдельных зон, питание для которых подавалось бы отдельно. На рис. 5.40 экран разбит штриховыми линиями на 16 зон. Когда курсор находится в окне 2, как показано на рис. 5.40, *а*, должны светиться только четыре зоны в правом нижнем углу. Остальные 12 зон могут оставаться темными, сохраняя, таким образом, 3/4 потребляемой экраном энергии.

Когда пользователь перемещает курсор на окно 1, зоны окна 2 могут быть затемнены, а зоны, на которых отображается окно 1, должны быть включены. Однако так как окно 1 накрывает 9 зон, требуется больше энергии. Если оконный менеджер знает о существовании энергетических зон экрана, он может автоматически переместить окно 1 так, чтобы оно помещалось в четыре зоны, как показано на рис. 5.40, *б*. Чтобы было возможно снизить потребление энергии с 9/16 полной мощности до 4/16 полной мощности, менеджер окон должен разбираться в вопросах

экономии электроэнергии или выполнять команды, получаемые от другой части операционной системы, занимающейся энергопотреблением. Еще более изощренной была бы возможность освещать частично окно, которое не содержит информации на всей своей площади, например, у окна, содержащего короткие строки текста, можно не освещать правую часть.

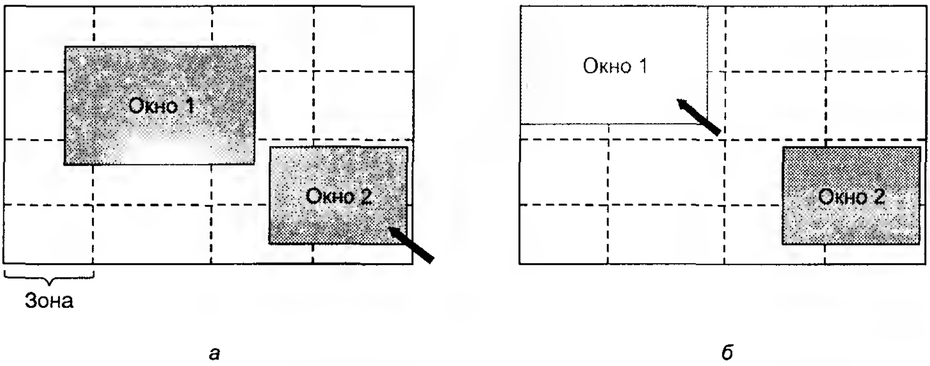


Рис. 5.40. Использование зон подсветки экрана: когда выбирается окно 2, оно не перемещается (*а*); когда выбирается окно 1, оно перемещается, чтобы уменьшить число подсвечиваемых зон (*б*)

Жесткий диск

Следующим главным «злодеем» является жесткий диск. Он потребляет значительное количество энергии для поддержания высокой скорости вращения даже при отсутствии обращений к нему. Многие компьютеры, особенно лэптопы, останавливают жесткий диск, если в течение определенного времени к нему нет обращений. Когда он оказывается нужен, диск запускается снова. К сожалению, время раскрутки диска занимает несколько секунд, что составляет весьма ощутимую для пользователя задержку.

Кроме того, на раскрутку диска требуется дополнительная энергия. В результате если диск останавливать и раскручивать слишком часто, то может оказаться, что энергетически выгоднее позволить диску вращаться постоянно, чем выключать и включать его снова. Каждый диск обладает характерным временем T_d, варьирующимся от 5 до 15 с. Предположим, что следующее обращение к диску ожидается через интервал времени t. Если $t < T_d$, то на поддержание вращения диска потребуется меньше энергии, чем на его последующую раскрутку. Если же $t > T_d$, то выгоднее окажется остановить диск, а затем запустить его снова. При способности хорошо предсказывать (на основе времени последних обращений к диску), когда произойдет следующее обращение к диску, операционная система могла бы оптимально экономить электроэнергию. На практике, однако, в большинстве систем применяется консервативная политика выключения устройств после истечения определенного интервала времени, в течение которого к ним не было обращений.

Другой способ сохранения энергии диска заключается в поддержании большого дискового кэша в оперативной памяти. Если нужный блок находится в памяти, обращения к диску при чтении этого блока не происходит, в результате чего диск можно не раскручивать. Аналогично, запись на диск также может буферизировать-

ся в кэше. При этом диск может оставаться выключенным, пока не переполнится кэш или не понадобится блок, отсутствующий в кэше[1].

Еще один способ избежать излишних запусков жесткого диска состоит в предоставлении операционной системой информации о состоянии диска работающей программе. Некоторые программы производят периодические операции записи на диск, которые можно пропустить или отложить. Например, текстовый процессор может быть настроен на сохранение редактируемого файла через определенные интервалы времени. Если текстовый процессор знает, что диск в данный момент выключен, он может отложить операцию сохранения до того момента, когда диск будет включен или просто подождать некоторое время.

Центральный процессор

Центральный процессор также может находиться в различных режимах энергопотребления. Центральный процессор лэптопа может быть программно переведен в режимах низкого энергопотребления, при котором он практически не потребляет энергии. В этом режиме он не выполняет никаких действий. Вывести из этого режима его может любой сигнал прерывания. Таким образом, если у центрального процессора нет текущих дел, он может перейти в режим низкого энергопотребления.

На многих компьютерах напряжение, потребляемое центральным процессором, его частота и потребляемая мощность связаны между собой. Напряжение питания центрального процессора часто можно уменьшить программно. При этом (приблизительно пропорционально) снизится его тактовая частота, зато (пропорционально квадрату напряжения) снизится энергопотребление. Например, снижение напряжения питания центрального процессора вдвое уменьшит его производительность также в два раза, зато энергии он будет потреблять в четыре раза меньше.

Это свойство может использоваться программами с хорошо известными сроками выполнения определенных операций, например программами просмотра мультимедиа, которым требуется через каждые 40 мс распаковать и отобразить один кадр. Однако если программе удается выполнить эту операцию быстрее, она может остаток временного интервала провести в режиме низкого энергопотребления процессора. Предположим, центральный процессор потребляет мощность, равную x Вт, при работе на полной мощности, и $x/4$ Вт при половинной скорости. Если программа может распаковать и отобразить кадр за 20 мс, операционная система может на остальные 20 мс полностью отключить процессор, в результате чего среднее потребление энергии процессором снизится вдвое. Однако операционная система может на все 40 мс снизить вдвое напряжение питания процессора, в результате чего процесс распаковки и вывода кадра на экран займет все 40 мс, зато энергии центральный процессор потребит меньше вчетверо (рис. 5.41). В обоих случаях выполняется один и тот же объем работ, но на рис. 5.41, *б* на это потребуется в два раза меньше энергии.

---

[1] При этом снизится надежность, так как на диске данные хранить все-таки надежнее, чем в памяти. — *Примеч. перев.*

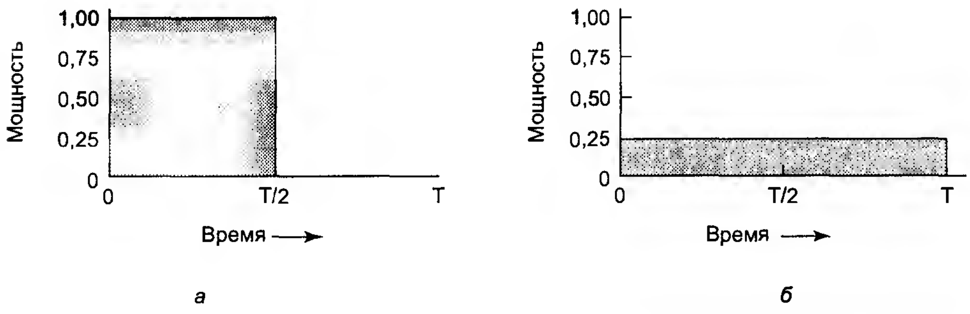


Рис. 5.41. Работа на полной скорости (*а*); снижение напряжения вдвое уменьшает в два раза частоту процессора, а потребляемую мощность вчетверо (*б*)

В схожей ситуации, если пользователь вводит один символ в секунду, а для обработки символа требуется 100 мс, операционная система может снизить скорость процессора в 10 раз. Подводя итоги, можно сказать, что медленная работа является более эффективной с энергетической точки зрения.

Оперативная память

Для энергосбережения при работе с памятью возможны два подхода. Во-первых, можно периодически сохранять и выключать кэш. Его всегда можно перезагрузить из оперативной памяти без потерь информации. Перезагрузка может выполняться автоматически и достаточно быстро, таким образом, данный вариант «сна» не является глубоким.

Более радикальный метод заключается в сохранении содержимого оперативной памяти на диске, после чего может быть выключена сама оперативная память. Этот метод представляет собой уже глубокую «спячку», так как перезагрузка может занять довольно существенное время, особенно если диск также останавливается. Когда память выключена, то и центральный процессор стоит выключить, так как обрабатывать ему будет практически нечего (кроме ПЗУ). При этом прерывание, активирующее центральный процессор, должно обрабатываться в ПЗУ, так как ОЗУ будет в этот момент выключено. Несмотря на довольно значительные накладные расходы по перезапуску, выключение оперативной памяти на длительный период времени (например, несколько часов) может быть оправдано, если перезапуск за несколько секунд оказывается предпочтительнее перезагрузки всей операционной системы с диска, что может занять около минуты или даже несколько минут.

Беспроводная связь

В последнее время появляется все большее число переносных компьютеров, соединенных с внешним миром (например, с Интернетом) при помощи беспроводной связи. Требующиеся для этого радиопередатчики и радиоприемники часто оказываются самыми мощными потребителями энергии. В частности, если радиоприемник находится в постоянно включенном состоянии, чтобы прослушивать поступающие сигналы, заряд аккумуляторов может иссякнуть довольно быстро. С другой стороны, если выключить радиоприемник после одной минуты простоя,

пришедшее сообщение может оказаться пропущенным, что, конечно, крайне нежелательно.

Одно эффективное решение этой проблемы было предложено в 1998 году Кравецом и Кришнаном [186]. В основу этого решения положен тот факт, что мобильные компьютеры общаются с неподвижными базовыми станциями, обладающими большими объемами оперативной памяти и дискового пространства, а также не связанными ограничениями в потреблении энергии. Их предложение заключается в том, что мобильный компьютер, собираясь выключить радио, передает базовой станции соответствующее сообщение. Получив его, базовая станция буферизирует все сообщения, адресованные мобильному компьютеру на своем диске. Спустя некоторое время, включив снова свою радиостанцию, мобильный компьютер первым делом сообщает об этом базовой станции. В ответ она присылает ему все сообщения, накопившиеся за это время.

Исходящие сообщения, сформированные в период, когда радио отключено, буферизируются на мобильном компьютере. Если буферу угрожает переполнение, радиопередатчик включается и сообщения, стоящие в очереди, передаются на базовую станцию.

Когда следует выключать радио? Можно разрешить принимать подобное решение пользователю или прикладной программе. Другой вариант состоит в отключении радиостанции через несколько секунд простоя. Когда следует снова включить радиопередатчик? Опять же, решение может принимать пользователь или прикладная программа, либо включение может производиться также периодически для проверки приходящих сообщений или для передачи сообщений, поставленных в очередь. Кроме того, передатчик должен включаться при угрозе переполнения буфера. Возможны также и другие решения.

Управление температурным режимом

Проблема управления температурным режимом несколько отличается от проблемы энергосбережения, тем не менее этот вопрос также имеет отношение к энергии. Из-за своей высокой частоты современные центральные процессоры очень сильно греются. Настольные машины обычно оснащаются внутренним электрическим вентилятором, сдувающим горячий воздух с шасси. Поскольку для настольных компьютеров снижение энергопотребления не является вопросом жизни и смерти, вентилятор обычно включен постоянно.

С лэптопами ситуация совершенно иная. Операционная система должна постоянно следить за температурой узлов компьютера. Когда температура начинает приближаться к максимально допустимому значению, у операционной системы есть выбор. Она может включить вентилятор, который шумит и потребляет энергию. В качестве альтернативы она может уменьшить энергопотребление, снизив освещенность экрана, уменьшив тактовую частоту процессора, чаще останавливая винчестер и т. д.

Пользователь также может принять участие в принятии решения. Например, он может заранее указать, что возражает против шума вентилятора, в результате операционной системе останется только снижать энергопотребление.

Управление состоянием батарей

Давным-давно батареи просто давали электрический ток, пока не иссякали, после чего их работа прекращалась. Сегодня все стало несколько сложнее и интереснее. В лэптопах сегодня применяются «умные» батареи, способные общаться с операционной системой. Получив запрос от операционной системы, они могут сообщить такие сведения, как максимальное напряжение, текущее напряжение, максимальный заряд, текущий заряд, максимальная скорость разрядки, текущая скорость разрядки и т. д. У большинства лэптопов имеются программы, которые можно запустить, чтобы отобразить все эти параметры. «Умные» батареи могут также изменять различные рабочие параметры под управлением операционной системы.

У некоторых лэптопов имеется несколько батарей. Когда операционная система обнаруживает, что одна из батарей скоро разрядится, она должна организовать корректное переключение на другую батарею. Когда же у последней батареи заканчивается заряд, операционная система должна предупредить пользователя и выполнить принудительное завершение работы, чтобы гарантировать, что файловая система не повреждена.

Интерфейс драйвера

В системе Windows имеется тщательно продуманный механизм управления энергопотреблением, называемый **ACPI** (Advanced System Configuration and Power Interface — усовершенствованный интерфейс конфигурирования системы и управления энергопотреблением). Операционная система может посылать любому драйверу, поддерживающему данный интерфейс, команды с требованием сообщить возможности его устройств и их текущие состояния. Эта способность особенно важна в комбинации с использованием стандарта Plug and Play, так как сразу после загрузки операционная система даже не знает, какие устройства присутствуют в компьютере, не говоря уже об их свойствах, касающихся энергопотребления или возможности управлять им.

Операционная система также может посылать драйверам команды, приказывая им снизить уровни энергопотребления (основываясь, естественно, на информации, полученной при загрузке).

Частичное функционирование

До сих пор мы рассматривали те способы, которыми операционная система может снизить потребление энергии различными видами устройств. Однако к решению данной проблемы есть и другой подход: велеть программе использовать меньше энергии, даже если это означает снижение уровня восприятия пользователя, называемое в данном контексте деградацией (лучше ухудшение восприятия, чем его полное отсутствие, наступающее после полного истощения аккумуляторов). Обычно программа получает такое сообщение, когда заряд батарей оказывается ниже определенного уровня. В этом случае прикладная программа может выбрать между снижением производительности для продления жизни батарей и сохранением производительности с риском нехватки энергии.

Таким образом, возникает вопрос: как может прикладная программа снизить свою производительность для экономии энергии? Этот вопрос был рассмотрен

Флинном и Сатьянарайянаном [117]. Они предложили четыре примера экономии энергии с помощью снижения производительности. Мы рассмотрим их ниже.

В данном исследовании информация предоставляется пользователю в различном виде. Когда снижения производительности нет, пользователю предоставляется максимально качественная информация. При снижении производительности качество воспроизведения (точность) информации, предоставляемой пользователю, оказывается хуже, чем это возможно. Мы кратко рассмотрим этот вопрос на примерах.

Для измерения используемой энергии Флинн и Сатьянарайянан разработали программный инструмент, названный ими PowerScope. Этот инструмент показывает мощность, потребляемую программой. Для его использования компьютер должен быть подключен к внешнему источнику питания через управляемый программно цифровой универсальный электроизмерительный прибор. При помощи этого прибора программное обеспечение может узнать силу тока, поступающего от источника питания, и таким образом определить мгновенную мощность, потребляемую компьютером. PowerScope периодически записывает в файл текущее состояние счетчика команд и соответствующее ему значение потребляемой мощности. По окончании работы программы этот файл анализируется, в результате чего можно узнать значения потребляемой мощности для каждой работающей на компьютере процедуры. Эти измерения образовали основу их наблюдений. Программные методы энергосбережения сравнивались с аппаратными, которые также применялись.

Первой программой, для которой были произведены измерения, стала программа воспроизведения видео. В нормальном режиме она воспроизводит 30 кадров в секунду в полном разрешении и цвете. Одна из форм снижения качества воспроизведения состоит в отображении видео в черно-белом изображении. Другая форма заключается в уменьшении частоты вывода кадров, что приводит к мерцанию и подергиванию изображения. Еще одна форма снижения качества воспроизведения заключается в уменьшении числа пикселов, выводимых на экран в каждом кадре, либо за счет уменьшения размеров отображаемого кадра, либо за счет снижения разрешения изображения. Экономия энергии составила в данном случае около 30 %.

Второй исследуемой программой был распознаватель речи. Эта программа сохраняла отсчеты сигнала микрофона. Полученный квантованный сигнал мог анализироваться либо на лэптопе, либо посылаться по радио для анализа стационарному компьютеру. При этом экономилась энергия, затрачиваемая центральным процессором, но расходовалась дополнительная энергия на радиосвязь. Деградация в данном случае представляла собой использование словаря меньшего размера и более простой акустической модели. В данном случае выигрыш составил около 35 %.

Следующим примером была программа просмотра карты, получающая карту по радио. Деградация заключалась либо в уменьшении размеров отображаемой карты, либо в том, что программа велела удаленному серверу опускать незначительные дороги, таким образом снижая количество битов, требуемых для передачи. Здесь также экономия составила около 35 %.

Четвертый эксперимент заключался в передаче JPEG-изображений web-браузеру. Стандартом JPEG поддерживаются различные алгоритмы, что позволяет выбирать между качеством изображения и размерами файла. В данном случае средний выигрыш составил всего 9 %. Тем не менее данные эксперименты показали, что, согласившись на некоторое снижение качества работы программы, пользователь может работать дольше с теми же аккумуляторами.

Исследования ввода-вывода

Проблемам ввода-вывода было посвящено много исследований, однако большая их часть была нацелена скорее на изучение конкретных устройств, нежели на ввод-вывод в целом.

Одним из рассматриваемых вопросов являются диски. В старых алгоритмах используется уже не применимая ныне модель диска, поэтому Уортингтоном и группой исследователей была рассмотрена модель, соответствующая современным дискам [363]. Система RAID в настоящее время находится в центре внимания исследователей, изучающих различные аспекты данной системы. Альварес с группой ученых занимались исследованием улучшения устойчивости к сбоям [10]. Этим же вопросом занимались Блаум и его коллеги [35]. Другие исследователи изучали идею использования в системе RAID параллельных контроллеров [51]. Третья группа исследователей описала прогрессивную систему RAID, которую они построили для Hewlett Packard [358]. Для большого количества дисков требуется тщательное параллельное планирование, что также исследовалось [64, 170]. Некоторые исследователи настаивают на использовании холостого времени после завершения операции поиска цилиндра, но до того, как нужный сектор окажется под головкой [219]. Еще более обещающими выглядят исследования электромеханических накопителей, не имеющих вращающихся деталей [136, 53], а также голографических накопителей [254]. Интересной новой технологией являются магнитооптические диски [229].

Идея использования SLIM-терминалов представляет собой современную версию старой системы разделения времени, в которой все вычисления производятся централизованно, а пользователям предоставляются терминалы, просто управляющие дисплеем, клавиатурой и мышью [295]. Основное отличие от старой системы разделения времени состоит в том, что вместо того, чтобы соединять терминал с компьютером через модем со скоростью 9600 бит/с, используется 10-мегабитная сеть Ethernet, обеспечивающая достаточную пропускную способность для полного графического интерфейса со стороны пользователя.

Графические интерфейсы пользователя стали в настоящее время практически стандартами, но тем не менее в этой сфере работы продолжаются, например в области речевого ввода [221, 222, 305, 335]. Внутренней структуре графического интерфейса пользователя также посвящены исследования [327].

При наличии огромного количества лэптопов у кибернетиков и очень короткого времени жизни батарей неудивительно, что проблема энергосбережения также является предметом интенсивных исследований [107, 117, 186, 198, 214, 217].

Резюме

Ввод-вывод, хотя им часто пренебрегают, является тем не менее важной темой. Существенная часть операционной системы занимается вводом-выводом. Операция ввода-вывода может выполняться тремя способами. Во-первых, при помощи программного ввода-вывода, при котором центральный процессор вводит или выводит каждый байт или слово, находясь в цикле ожидания готовности устройства ввода-вывода. Второй способ представляет собой управляемый прерываниями ввод-вывод, при котором центральный процессор начинает передачу ввода-вывода для символа или слова, после чего переключается на другой процесс, пока прерывание от устройства не сообщит ему об окончании операции ввода-вывода. Третий способ заключается в использовании прямого доступа к памяти (DMA), при котором отдельная микросхема управляет переносом целого блока данных, и инициирует прерывание только после окончании операции переноса блока.

Ввод-вывод можно разбить на четыре уровня иерархии: процедуры обработки прерываний, драйверы устройств, независимое от устройств программное обеспечение ввода-вывода и библиотеки ввода-вывода и спулеры, работающие в пространстве пользователя.

Драйверы устройств управляют деталями работы устройств и предоставляют однородные интерфейсы к остальной части операционной системы. Независимое от устройств программное обеспечение ввода-вывода занимается буферизацией и сообщением об ошибках.

Существует большое количество типов дисков, включая магнитные диски, системы RAID и различные типы оптических дисков. Алгоритмы планирования перемещения блока головок часто могут улучшить производительность диска, но наличие виртуальной геометрии усложняет эту задачу. При помощи объединения двух дисков в пару может быть создан надежный носитель данных с определенными полезными свойствами.

Часы используются для определения текущего значения реального времени, ограничения времени работы процессов, управления сторожевыми таймерами, а также сбора статистической информации.

Алфавитно-цифровые терминалы позволяют вводить специальные управляющие символы с клавиатуры, а вывод на дисплей алфавитно-цифрового терминала управляется ESC-последовательностями. Драйвер клавиатуры может работать как в «сыром» режиме, так и в режиме с обработкой, в зависимости от потребностей конкретной прикладной программы. ESC-последовательности при выводе на терминал управляют перемещением курсора, а также позволяют вводить и удалять текст с экрана.

Многие персональные компьютеры используют для вывода графические интерфейсы пользователя, основывающиеся на WIMP-парадигме: окнах, пиктограммах, меню и указывающем устройстве. Программы, пользующиеся графическим интерфейсом пользователя, обычно управляются событиями. Ввод с клавиатуры, мыши, а также другие события посылаются на обработку программе.

Существует несколько типов сетевых терминалов. Один из наиболее популярных сетевых терминалов работает под управлением оконной системы X Windows,

на основе которой могут быть построены различные варианты графических интерфейсов пользователя. В качестве альтернативы системе X Windows может применяться низкоуровневый интерфейс, просто переносящий необработанные пикселы по сети. Эксперименты со SLIM-терминалами доказали удивительно хорошую работоспособность такого метода.

Наконец, управление энергопотреблением является вопросом номер один для портативных компьютеров, так как время работы аккумуляторов сильно ограничено. Операционной системой могут применяться различные методы снижения энергопотребления. Кроме того, возможен программный метод экономии энергии за счет некоторого снижения качества работы.

Вопросы

1. Благодаря прогрессу в сфере технологии производства микросхем стало возможным поместить в одну недорогую микросхему весь контроллер, включая всю логику доступа к шине. Как это повлияло на модель, изображенную на рис. 1.5?
2. Основываясь на скоростях, перечисленных в табл. 5.1, скажите, возможно ли сканировать на полной скорости документы со сканера на EIDE-диск, подключенный к шине ISA? Обоснуйте свой ответ.
3. На рис. 5.2, *б* показан один из способов работы с отображаемыми на адресное пространство памяти устройствами ввода-вывода, даже при наличии отдельных шин для памяти и устройств ввода-вывода. Этот способ состоит в том, что сначала проверяется шина памяти, а в случае неудачи обращение производится к шине ввода-вывода. Умный студент факультета технической кибернетики решил усовершенствовать эту модель, предложив распараллелить обращения по двум шинам, чтобы ускорить доступ к устройствам ввода-вывода. Что вы думаете о его идее?
4. У контроллера DMA четыре канала. Контроллер способен запрашивать 32-разрядное слово через каждые 100 нс. Ответ на запрос занимает столько же времени. Насколько быстрой должна быть шина, чтобы не стать узким местом системы?
5. Предположим, что компьютер может обращаться к памяти с операциями чтения или записи слова за 10 нс. Также предположим, что при прерывании все 32 регистра центрального процессора плюс счетчик и PSW сохраняются в стеке. Какое максимальное количество прерываний в секунду может обработать эта машина?
6. На рис. 5.6, *б* прерывание не подтверждается до тех пор, пока новый символ не будет выведен на принтер. Можно ли было подтвердить прерывание в самом начале процедуры обработки прерываний? Если да, то укажите причину, по которой это действие выполняется в конце, как показано в тексте. Если нет, объясните, почему.

7. У компьютера есть трехуровневый конвейер, как показано на рис. 1.6, *а*. На каждом такте процессор выбирает из памяти одну команду с адреса, на который указывает счетчик команд, и помещает ее в конвейер, после чего значение счетчика команд увеличивается. Предположим, каждая команда занимает ровно одно слово процессора. Команды, находящиеся в конвейере, продвигаются вперед на одно слово. При прерывании текущее состояние счетчика команд сохраняется в стеке, а в счетчик команд заносится адрес обработчика прерываний. Затем конвейер смещается на один шаг вправо, и первая команда обработчика прерываний попадает в конвейер. Обладает ли такая машина точными прерываниями? Обоснуйте свой ответ.
8. Типичная печатная страница текста состоит из 50 строк по 80 символов в каждой. Представьте себе принтер, способный печатать 6 страниц в минуту, причем время вывода на принтер одного символа настолько мало, что им можно пренебречь. Имеет ли смысл управлять выводом на этот принтер при помощи прерываний, если для печати каждого символа требуется прерывание, занимающее 50 мкс?
9. Что такое независимость от устройств?
10. В каком из четырех уровней программного обеспечения ввода-вывода выполняются следующие действия:
    а) вычисление номеров дорожки, сектора и головки для чтения диска;
    б) запись команд в регистры устройства;
    в) проверка разрешения доступа пользователя к устройству;
    г) преобразование двоичного целого числа в ASCII-символы для вывода на печать.
11. Основываясь на данных табл. 5.3, скажите, чему равна скорость переноса данных между диском и контроллером для гибкого диска и жесткого диска? Сравните полученные результаты с 56-килобитным модемом и 100-мегабитной быстрой сетью Ethernet.
12. Локальная сеть используется следующим образом. Пользователь обращается к системному вызову, чтобы записать пакеты данных в сеть. Затем операционная система копирует данные в буфер ядра. После этого данные копируются в плату сетевого контроллера. После того как все байты попадают в контроллер, они посылаются по сети со скоростью 10 Мбит/с. Получающий сетевой контроллер сохраняет каждый бит спустя 1 мкс после его отправки. Когда последний бит получен, центральный процессор получающего компьютера прерывается и ядро копирует прибывший пакет в свой буфер, чтобы исследовать его. Поняв, какому пользователю предназначается пакет, ядро копирует данные в пространство пользователя. Если предположить, что каждое прерывание и его обработка занимает 1 мс, размер пакетов равен 1024 байт (не считая заголовков), а копирование одного байта занимает 1 мкс, то чему будет равна максимальная скорость, с которой один процесс может передавать данные другому процессу? Предположите, что отправи-

тель блокируется, пока получатель не закончит работу и не отправит обратно подтверждение. Для простоты предположим, что временем получения подтверждения можно пренебречь.

13. Почему файлы, посылаемые на принтер, обычно перед печатью накапливаются на диске?
14. Чему должен быть равен перекос цилиндров для диска, вращающегося со скоростью 7200 об/мин, при времени перемещения головки на соседнюю дорожку, равном 1 мс? У диска по 200 секторов по 512 байт на каждой дорожке.
15. Сосчитайте максимальную скорость передачи данных в мегабайтах в секунду для диска из предыдущего вопроса.
16. Система RAID уровня 3 способна исправлять однобитовые ошибки при помощи только одного диска четности. В чем суть системы RAID уровня 2? В конце концов, она также может исправлять только однократные ошибки и требует большее число дисков для этого.
17. Система RAID может не справиться со своей задачей, если в течение небольшого интервала времени из строя выйдут сразу два или более дисков. Предположим, что вероятность выхода одного диска из строя в течение одного часа равна p. Чему равна вероятность выхода из строя системы RAID, состоящей из k дисков в течение одного часа?
18. Почему оптические устройства хранения данных обладают более высокой плотностью записи данных, чем магнитные накопители? *Замечание*: этот вопрос требует некоторых знаний физики, в частности способа формирования магнитных полей.
19. Если контроллер диска записывает получаемые от диска байты в память так же быстро, как и получает их, без внутреннего буферирования, будет ли польза от чередования секторов? Обоснуйте.
20. Представьте себе гибкий диск, у которого шаг чередования секторов равен двум, как на рис. 5.22, *в*. На каждой дорожке диска восемь секторов по 512 байт. Скорость вращения диска 300 об/мин. Сколько времени потребуется, чтобы прочитать все секторы дорожки в правильном порядке, если предположить, что головка диска уже стоит на нужной дорожке, а чтобы сектор 0 переместился под головку, требуется 1/2 оборота диска? Чему равна скорость считывания данных? Теперь повторите то же задание для диска без чередования с теми же характеристиками. Как сильно снижается скорость из-за чередования?
21. Если на диске применяется двукратное чередование, то требуется ли еще применять перекос цилиндров во избежание потерь данных при переходе с дорожки на дорожку? Обоснуйте свой ответ.
22. Фирма выпускает два 5-дюймовых жестких диска, у каждого из которых 10 000 цилиндров. Новый жесткий диск имеет двойную линейную плотность записи по сравнению с более старым диском. Какие свойства у нового диска будут лучше, чем у старого, а какие останутся такими же?

23. Производитель компьютеров решает изменить таблицу разделов на жестком диске Pentium-компьютера, чтобы предоставить возможность создания более четырех разделов. Каковы будут последствия этих изменений?

24. Драйвер диска получает запросы на чтение/запись к цилиндрам 10, 22, 20, 2, 40, 6 и 38. Перемещение блока головок с одного цилиндра на соседний занимает 6 мс. Сколько потребуется времени на перемещение головок при использовании алгоритма:

    а) обслуживания в порядке поступления запросов;

    б) обслуживания в первую очередь ближайшего цилиндра;

    в) элеваторного алгоритма (сначала блок головок двигается вверх).

    Во всех случаях начальное положение блока головок на цилиндре 20.

25. Продавец персональных компьютеров, посещая университет в Юго-Западном Амстердаме (на юго-западе Амстердама?) для продажи партии компьютеров, заявляет, что его компания приложила существенные усилия по ускорению их версии UNIX. В качестве примера он отмечает, что в их драйвере диска применяется элеваторный алгоритм, а также обслуживание очереди запросов к одному цилиндру в порядке секторов. На студента Гарри Хакера эго речь производит настолько сильное впечатление, что он покупает один компьютер. Гарри приносит компьютер домой и пишет программу, читающую случайные 10 000 блоков диска. К его изумлению, замеренная им производительность идентична той, которой можно было ожидать при использовании алгоритма обслуживания запросов в порядке поступления. Означает ли это, что продавец лгал?

26. В обсуждении темы стабильного запоминающего устройства, использующего энергонезависимое ОЗУ, был обойден вниманием следующий момент. Что произойдет, если операция стабильной записи будет выполнена, но случится сбой, прежде чем операционная система сможет записать номер неправильного блока в энергонезависимое ОЗУ? Разрушает ли подобное состояние состязаний абстракцию стабильного запоминающего устройства? Поясните свой ответ.

27. На некотором компьютере обработчик прерываний от таймера выполняет свои действия за 2 мс (включая накладные расходы по переключению процессов). Прерывания от таймера поступают с частотой 60 Гц. Какая часть времени работы центрального процессора расходуется на таймер?

28. Во многих версиях UNIX значение времени хранится в 32-разрядном целом числе, в виде числа секунд от некоторой точки отсчета. Когда перестанет хватать 32-разрядного числа для хранения значения времени (год и месяц)?

29. Некоторым компьютерам, например Интернет-серверам, приходится поддерживать большое количество линий RS-232. Для этого существуют платы, поддерживающие несколько линий RS-232. Допустим, такая плата содержит процессор, проверяющий состояние каждой входной линии (0 или 1), с частотой, в 8 раз большей скорости линии в Бодах. Допустим, что такое измере-

ние состояния линии занимает 1 мкс. Сколько линий, работающих со скоростью 3200 бод и способных передавать данные со скоростью 28 000 бит/с, может поддержать процессор? Примечание: скорость линии в бодах равна количеству изменений сигнала в секунду. Линия со скоростью 3200 бод может передавать данные со скоростью 28 000 бит/с, если каждый сигнал переносит 9 бит, используя различные амплитуды, частоты и фазы. Следует отметить, что модемы, работающие со скоростью 56 кбит/с, не используют RS-232.

30. Почему терминалы, использующие интерфейс RS-232, управляются прерываниями, а терминалы с отображением на память — нет?

31. Рассмотрим производительность модема со скоростью 56 кбит/с. Драйвер посылает один символ, после чего блокируется. После того как символ передан в линию, происходит прерывание, после чего драйвер разблокируется и посылает следующий символ и т. д. Какую часть времени центрального процессора занимает управление модемом, если обработка прерывания, вывод одного символа и блокировка занимают 100 мкс? Предположите, что у каждого передаваемого модемом символа имеется один стартовый и один стоповый бит, а всего символ занимает 10 бит.

32. Растровый терминал содержит 1280×960 пикселов. Для скроллинга окна центральный процессор (или контроллер) должен переместить все строки текста вверх, копируя их биты из одной части видеопамяти в другую. Допустим, в окне 60 строк по 80 символов в строке (всего 5280 символов), а каждый символ имеет 8 пикселов в ширину и 16 пикселов в высоту. Сколько времени займет скроллинг всего окна, если для копирования одного байта требуется 50 нс? Если все строки имеют по 80 символов в длину, чему будет равна эквивалентная скорость терминала в бодах? Помещение одного символа на экран занимает 5 мкс. Сколько строк в секунду может быть выведено в окно?

33. Получив символ DEL (SIGINT), драйвер дисплея очищает всю очередь на вывод для этого дисплея. Почему?

34. Пользователь терминала, использующего интерфейс RS-232, дает редактору команду удалить слово в строке 5, занимающее позиции с 7 по 12 включительно. Какую ESC-последовательность стандарта ANSI должен выдать редактор, чтобы удалить слово, если предположить, что курсор не находится на строке 5 в момент подачи команды?

35. У многих терминалов, использующих интерфейс RS-232, есть ESC-последовательности для удаления текущей строки и перемещения всех нижних строк на одну строку вверх. Как, по-вашему, реализована эта операция внутри терминала?

36. На оригинальном компьютере IBM PC с цветным дисплеем запись в видеопамять в любое время, кроме того интервала, когда электронный луч совершал вертикальный обратный ход, вызывала появление уродливых пятен по всему экрану. На экран выводится 25 строк по 80 символов, каждый из которых помещается в квадрат 8×8 пикселов. Каждый ряд из 640 пикселов

рисуется за один горизонтальный проход луча, что занимает 63,6 мкс, включая горизонтальный обратный ход луча. Экран перерисовывается по 60 раз в секунду. При каждом выводе экрана требуется период времени на вертикальный обратный ход луча. Какую часть времени видеопамять оказывается доступной для записи?

37. Разработчики компьютерной системы предполагали, что максимальная скорость перемещения мыши составит 20 см/с. Если один мики равен 0,1 мм, а каждое сообщение мыши занимает три байта, чему будет равна максимальная скорость передачи данных, при условии, что о каждом мики сообщается отдельно?
38. Основными дополнительными цветами являются красный, зеленый и синий. Это означает, что любой цвет может быть сконструирован как линейная суперпозиция этих цветов. Может ли существовать такая цветная фотография, которую невозможно представить при помощи 24-разрядного цвета?
39. Один из способов вывода символа на растровый экран состоит в копировании его из таблицы шрифтов процедурой bitblt. Предположим, что некоего шрифта используются символы размером 16×24 пиксела в формате RGB.
    а) Сколько места займет каждый символ в таблице шрифта?
    б) Если копирование одного байта занимает 100 нс, включая накладные расходы, чему равна скорость вывода в символах в секунду?
40. Предполагая, что для копирования одного байта требуется 10 нс, сколько времени понадобится, чтобы полностью перерисовать отображаемый на адресное пространство памяти экран, работающий в символьном режиме с разрешением 25 строк по 80 символов? Какой результат получится в графическом режиме с разрешением 1024×768 пикселов при 24 битах на пиксел?
41. В листинге 5.2 есть обращение к процедуре *RegisterClass* для регистрации объекта *wndclass*. В соответствующей программе для X Windows в листинге 5.3 нет ничего похожего на этот вызов. Почему?
42. В тексте был приведен пример вывода на экран прямоугольника при помощи интерфейса Windows GDI:

    ```
    Rectangle(hdc, xleft, ytop, xright, ybottom);
    ```

    Действительно ли здесь нужен первый параметр (*hdc*), и если да, то зачем? В конце концов, координаты прямоугольника явно задаются остальными параметрами.
43. SLIM-терминал используется для отображения web-страницы, содержащей мультфильм с размерами 400×160 пикселов, выводящийся на экран с частотой 10 кадров в секунду. Какую часть пропускной способности 100-мегабитной быстрой сети Ethernet занимает передача по сети мультфильма?
44. Тест показал хорошую работу SLIM-терминала с 1-мегабитной сетью. Могут ли возникнуть проблемы в многопользовательской ситуации? *Подсказка*: представьте большое количество пользователей, смотрящих запланированную телевизионную передачу, и то же число пользователей, просматривающих в браузере сайты Интернета.

45. При снижении напряжения питания центрального процессора в n раз во столько же раз уменьшается его тактовая частота, а энергопотребление снижается в n^2 раз. Предположим, что пользователь печатает один символ в секунду, но на обработку введенного символа центральному процессору требуется всего 100 мс. Чему будет равно оптимальное значение n и сколько энергии удастся сэкономить таким образом? Предполагается, что простаивающий центральный процессор вообще не потребляет энергии.
46. Лэптоп максимально использует режим энергосбережения, включая выключение дисплея и жесткого диска, когда они не используются в течение некоторого времени. Пользователь иногда запускает UNIX-программы в текстовом режиме, а временами использует систему X Windows. Пользователь замечает, что аккумуляторов хватает намного дольше при использовании текстовых программ. Почему?
47. Напишите программу, имитирующую стабильное запоминающее устройство. Используйте для моделирования двух дисков два файла фиксированной длины.

Глава 6
Файловые системы

Всем компьютерным приложениям нужно хранить и получать информацию. Во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве. Однако емкость такого хранилища ограничена размерами виртуального адресного пространства. Для некоторых приложений такого размера вполне достаточно, но для других, например систем резервирования авиабилетов, систем банковского или корпоративного учета, одного только виртуального адресного пространства будет недостаточно.

Кроме того, после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. Для большинства приложений (например, баз данных) эта информация должна храниться неделями, месяцами или даже вечно. Исчезновение данных после завершения работы процесса для таких приложений неприемлемо. Информация должна сохраняться даже при аварийном завершении процесса в случае сбоя компьютера.

Третья проблема состоит в том, что часто возникает необходимость нескольким процессам одновременно получить доступ к одним и тем же данным (или части данных). Если интерактивный телефонный справочник будет храниться в адресном пространстве одного процесса, то доступ к нему будет только у этого процесса. Для решения этой проблемы необходимо отделить информацию от процесса.

Таким образом, к долговременным устройствам хранения информации предъявляются три следующих важных требования:

1. Устройства должны позволять хранить очень большие объемы данных.
2. Информация должна сохраняться после прекращения работы процесса, использующего ее.
3. Несколько процессов должны иметь возможность получения одновременного доступа к информации.

Обычное решение всех этих проблем состоит в хранении информации на дисках и других внешних хранителях в модулях, называемых **файлами**. Процессы могут по мере надобности читать их и создавать новые файлы. Информация, хранящаяся в файлах, должна обладать **устойчивостью** (в данном контексте иногда применяется термин **персистентность**), то есть на нее не должны оказывать влияния создание или прекращение работы какого-либо процесса. Файл должен исчезать только тогда, когда его владелец дает команду удаления файла.

Файлами управляет операционная система. Их структура, именование, использование, защита, реализация и доступ к ним являются важными пунктами устрой-

ства операционной системы. Часть операционной системы, работающая с файлами, называется **файловой системой**. Ей и посвящена данная глава.

С точки зрения пользователя наиболее важным аспектом файловой системы является ее внешнее представление, то есть именование и защита файлов, операции с файлами и т. д. Такие детали внутреннего устройства, как использование связанных списков или бит-карт для слежения за свободными и занятыми блоками диска, число физических секторов в логическом блоке, представляют для пользователя меньший интерес, хотя и крайне важны для разработчиков файловой системы. По этой причине мы разбили главу на несколько разделов. Первые два раздела посвящены пользовательскому интерфейсу файлов и каталогов. В следующих разделах мы рассмотрим способы реализации файловой системы. Наконец, будут приведены несколько примеров существующих файловых систем.

Файлы

В следующих нескольких разделах мы рассмотрим файлы с точки зрения пользователя, то есть обсудим их использование и их свойства.

Именование файлов

Файлы относятся к абстрактному механизму. Они предоставляют способ сохранять информацию на диске и считывать ее снова позднее. При этом от пользователя должны скрываться такие детали, как способ и место хранения информации, а также детали работы дисков.

Вероятно, наиболее важной характеристикой любого механизма абстракции является то, как именуются управляемые объекты, поэтому мы начнем изучение файловой системы с именования файлов. При создании файла процесс дает файлу имя. Когда процесс завершает работу, файл продолжает свое существование и по его имени к нему могут получить доступ другие процессы.

Точные правила именования файлов варьируются от системы к системе, но все современные операционные системы поддерживают использование в качестве имен файлов 8-символьные текстовые строки. Таким образом, *andrea*, *bruce* и *cathy* являются допустимыми именами файлов. Часто в именах файлов также разрешается использование цифр и специальных символов, поэтому могут применяться и такие имена файлов, как *2*, *urgent!* и *Fig.2-14*. Многие файловые системы поддерживают имена файлов длиной до 255 символов.

В некоторых файловых системах, например UNIX, различаются прописные и строчные символы, тогда как в других, таких как MS-DOS, нет. Таким образом, имена файлов *maria*, *Maria* и *MARIA* будут означать в системе UNIX три различных файла, тогда как в MS-DOS все эти имена будут соответствовать одному файлу.

Операционные системы Windows 95 и Windows 98 используют файловую систему MS-DOS и наследуют многие ее свойства, включая именование файлов. Операционные системы Windows NT и Windows 2000 также поддерживают файловую систему MS-DOS и наследуют ее свойства. Однако у последних двух операционных систем имеется своя файловая система (NTFS), обладающая отличными

свойствами (например, именами файлов в кодировке Unicode). В данной главе при упоминании файловой системы Windows мы будем иметь в виду файловую систему MS-DOS, являющуюся единственной файловой системой, поддерживаемой всеми версиями Windows. Файловая система NTFS, используемая в Windows 2000, будет обсуждаться в главе 11.

Во многих операционных системах имя файла может состоять из двух частей, разделенных точкой, например *prog.c*. Часть имени файла после точки называется **расширением файла** и обычно означает тип файла. Так, в MS-DOS имя файла может содержать от 1 до 8 символов плюс расширение от 0 до 3 символов. В системе UNIX размер расширения файла зависит от пользователя. Кроме того, у файла может быть несколько расширений, например *prog.c.Z*, где *.Z* обычно используется, чтобы указать, что файл (*prog.c*) был сжат с помощью алгоритма Зива—Лемпеля. Некоторые часто встречающиеся расширения файлов и их значения приведены в табл. 6.1.

Таблица 6.1. Некоторые типичные расширения файлов

Расширение	Значение
file.bak	Резервная копия файла
file.c	Исходный текст программы на С
file.gif	Изображение формата GIF
file.hlp	Файл справки
file.html	Документ в формате HTML (web-страница)
file.jpg	Неподвижное изображение стандарта JPEG
file.mp3	Музыка в формате MPEG-1 уровень 3
file.mpg	Фильм в формате MPEG
file.o	Объектный файл (еще не скомпонованный выходной файл компилятора)
file.pdf	Документ формата PDF (программы Adobe Acrobat)
file.ps	Документ формата PostScript
file.tex	Входной файл для программы форматирования TEX
file.txt	Текстовый файл общего назначения
file.zip	Архив, сжатый с помощью алгоритма Зива—Лемпеля

В некоторых системах (например, в UNIX) расширения файлов являются просто соглашениями, и операционная система не принуждает пользователя их строго придерживаться. Файл *file.txt* может быть текстовым файлом, но это скорее напоминание пользователю, а не руководство к действию для операционной системы. С другой стороны, компилятор языка С может отказаться компилировать файлы с расширениями, отличными от *.c*.

Подобные соглашения особенно полезны, когда одна и та же программа должна управлять различными типами файлов. Например, компилятору языка С может быть предоставлен список файлов, которые он должен откомпилировать и скомпоновать, причем некоторые из этих файлов могут содержать программы на языке С, тогда как другие являться ассемблерными файлами. В этом случае именно по расширениям файлов компилятор сможет отличить одни файлы от других.

Система Windows, напротив, знает о расширениях файлов и назначает каждому расширению определенное значение. Пользователи (или процессы) могут регистрировать расширения в операционной системе, указывая программу, «владеющую» данным расширением. При двойном щелчке мыши на имени файла запускается программа, назначенная этому расширению, с именем файла в качестве параметра. Например, двойной щелчок мыши на имени *file.doc* запускает Microsoft Word, который открывает файл *file.doc*.

Структура файла

Файлы могут быть структурированы несколькими различными способами. Три типа структур показаны на рис. 5.1. Файл на рис. 5.1, *а* представляет собой неструктурированную последовательность байтов. В этом случае операционная система не интересуется содержимым файла. Все, что она видит — это байты. Значения этим байтам придается программами уровня пользователя. Такой подход используется как в системе UNIX, так и в Windows.

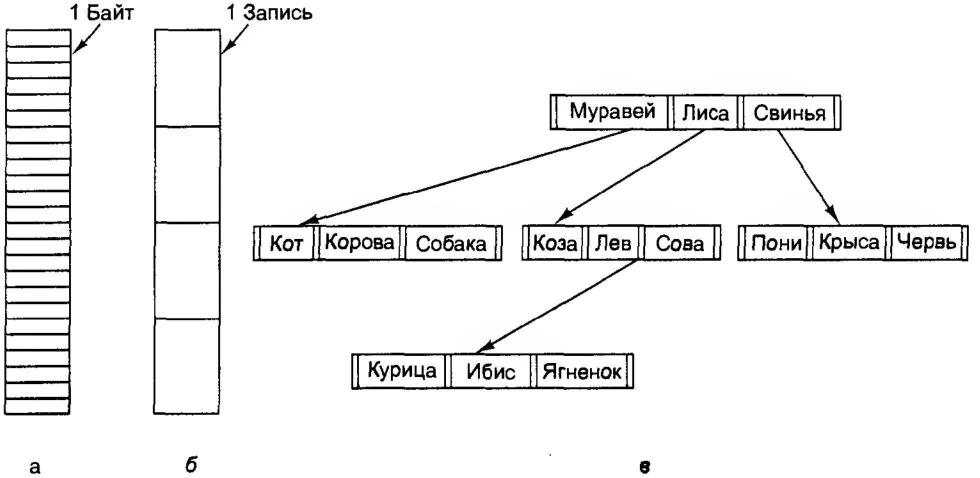


Рис. 6.1. Три типа файлов: последовательность байтов (а); последовательность записей (*б*); дерево (*в*)

Рассмотрение операционной системой файлов как просто последовательностей байтов обеспечивает максимальную гибкость. Программы пользователя могут помещать в файлы все что угодно и именовать их любым удобным для них способом. Операционная система не вмешивается в этот процесс, что может быть особенно ценно для пользователей, собирающихся сделать что-либо необычное.

Первый шаг по направлению к структуре показан на рис. 6.1, *б*. В данной модели файл представляет собой последовательность записей фиксированной длины, каждая со своей внутренней структурой. Для файлов, состоящих из записей, важным является то, что операция чтения возвращает одну запись, а операция записи перезаписывает или дополняет одну запись. Несколько десятилетий назад, когда вовсю применялись перфокарты, состоящие из 80 колонок отверстий, многие операционные системы (на мэйнфреймах) оперировали файлами, состоящими

из 80-символьных записей, представляющими собой образы перфокарт. Этими операционными системами поддерживались также файлы, состоящие из 132-символьных записей, предназначающихся для строковых принтеров (которые в те дни печатали по 132 символа в строке). Программы читали из входных файлов 80-символьные блоки и записывали их в виде 132-символьных блоков, хотя остальные 52 символа могут быть пробелами. Ни одна современная универсальная система не работает подобным образом.

Третий вариант файловой структуры показан на рис. 6.1, *в*. При такой организации файл представляет собой дерево записей, не обязательно одной и той же длины. Каждая запись в фиксированной позиции содержит поле **ключа**. Дерево сортировано по ключевому полю, что обеспечивает быстрый поиск заданного ключа.

Основной файловой операцией здесь является не получение следующей записи, хотя это также возможно, а получение записи с указанным значением ключа. Для файла зоопарка, показанного на рис. 6.1, *в*, можно, например, запросить у системы запись с ключом *пони*, не беспокоясь о точном положении этой записи в файле. При добавлении новых записей операционная система, а не пользователь должна решать, куда ее поместить. Такой тип файлов принципиально отличается от неструктурированных потоков байтов, применяемых в UNIX и Windows, но они широко применяются на больших мэйнфреймах, еще используемых для коммерческой обработки данных.

Типы файлов

Многие операционные системы поддерживают различные типы файлов. Например, в системах UNIX и Windows проводится различие между регулярными (обычными) файлами и каталогами. В системе UNIX также различаются символьные и блочные специальные файлы. К **регулярным** файлам относятся все файлы, содержащие информацию пользователя. Все файлы на рис. 6.1 являются регулярными. **Каталоги** — это системные файлы, обеспечивающие поддержку структуры файловой системы. Мы рассмотрим их подробнее ниже. **Символьные специальные файлы** имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, таких как терминалы, принтеры и сети. **Блочные специальные файлы** используются для моделирования дисков. В данной главе мы в первую очередь будем рассматривать регулярные файлы.

Регулярные файлы в основном являются либо ASCII-файлами, либо двоичными файлами. ASCII-файлы состоят из текстовых строк. В некоторых системах каждая строка завершается символом возврата каретки. В других (например, UNIX) используется символ перевода строки. В некоторых системах (например, MS-DOS) используются оба символа. Строки не обязаны иметь одну и ту же длину.

Большим преимуществом ASCII-файлов является то, что они могут отображаться на экране и выводиться на печать так, как есть, без какого-либо преобразования, и могут редактироваться практически любым текстовым редактором. Более того, если большое количество программ использует ASCII-файлы для входа и выхода, то оказывается несложным соединить вход одной программы с выходом другой, как это делается в конвейерах оболочки. (Обмен данными между процессами при этом не становится проще, но интерпретация информации облегчается, если для ее выражения применяется стандарт, такой как ASCII.)

Остальные файлы называются двоичными, то есть они не являются ASCII-файлами. При выводе их на принтер получается невразумительный набор символов, напоминающий случайный мусор. Обычно у них есть некая внутренняя структура, известная программе, использующей их.

Например, на рис. 6.2, *а* показан простой исполняемый двоичный файл одной из версий системы UNIX. Хотя технически файл представляет собой всего лишь последовательность байтов, операционная система станет исполнять файл только в том случае, если этот файл имеет соответствующий формат. Файл состоит из пяти разделов: заголовка, текста, данных, релокационных битов и таблицы символов. Заголовок начинается с так называемого **«магического» числа**, идентифицирующего файл как исполняемый (чтобы предотвратить случайное исполнение файла другого формата). Следом за «магическим» числом в заголовке располагаются размеры различных частей файла, адрес начала исполнения файла и некоторые флаговые биты. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и настраиваются на работу по адресу загрузки при помощи битов релокации. Таблица символов используется для отладки.

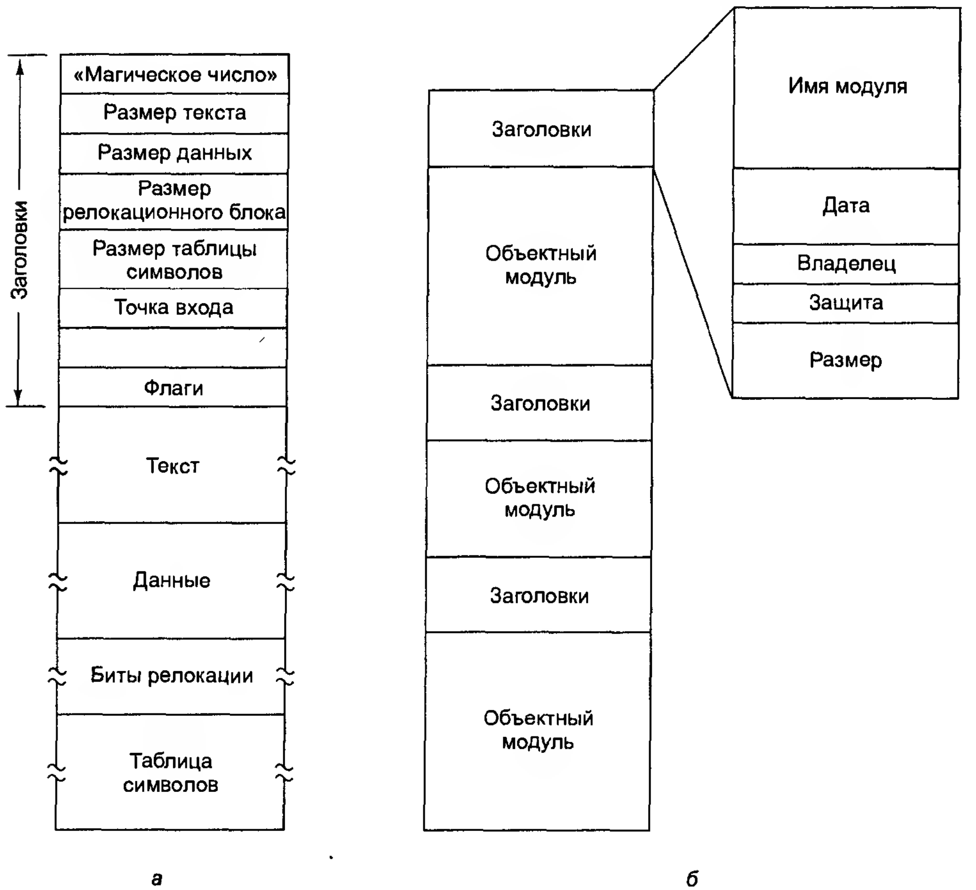


Рис. 6.2. Исполняемый файл (*а*); архив (*б*)

Второй пример двоичного файла представляет собой файл архива, также из системы UNIX. Он состоит из набора библиотечных процедур (модулей), откомпилированных, но не скомпонованных. Каждой процедуре предшествует заголовок, содержащий ее имя, дату создания, владельца, код защиты и размер. Как и в случае исполняемого файла, заголовки модулей содержат большое количество двоичных чисел. Если скопировать их на принтер, получится полная тарабарщина.

Все операционные системы должны распознавать по крайней мере один тип файлов — свои собственные исполняемые файлы, но некоторые операционные системы распознают и другие типы файлов. Старая система TOPS-20 (для компьютера DECsystem 20) даже изучала время создания каждого предоставляемого ей на исполнение файла. Затем она находила исходный файл и проверяла, не был ли он изменен, после того как был создан исполняемый файл. Если оказывалось, что исполняемый файл уже устарел, операционная система автоматически перекомпилировала исходный файл. Если перевести это на язык понятий UNIX, то это означало, что программа *make* была встроена в оболочку. Расширения файлов были обязательными, чтобы операционная система могла определить, какая двоичная программа от какого исходного файла произошла.

Однако такая жесткая заданность типов файлов может оказаться неудобной для пользователя, пытающегося сделать что-нибудь, не предусмотренное проектировщиками операционной системы. Представьте, например, систему, в которой файлы программного вывода автоматически получают расширение *.dat* (файлы данных). Пусть пользователь написал программу, форматирующую исходные тексты программ на С. Эта программа читает файл с расширением *.c*, обрабатывает его и затем пишет результат в файл со стандартным расширением *.dat*. Если пользователь затем попытается предложить этот файл С-компилятору, операционная система не даст этого сделать, так как у файла для данного действия неверное расширение. Попытка скопировать *file.dat* в *file.c* также будет отвергнута операционной системой.

Хотя такая «дружественность» по отношению к пользователю (защищающая пользователя от ошибок) может быть полезна для новичков, она ставит опытных пользователей в безвыходное положение, заставляя их тратить массу усилий на попытки перехитрить операционную систему.

Доступ к файлам

В старых операционных системах предоставлялся только один тип доступа к файлам — **последовательный доступ**. В этих системах процесс мог читать байты или записи файла только по порядку от начала к концу. Такой доступ к файлам появился, когда дисков еще не было и компьютеры оснащались магнитофонами. Поэтому даже в дисковых операционных системах при последовательном доступе к файлу имитировалось его чтение или запись на накопителе на магнитной ленте с возможностью многократной перемотки назад.

С появлением дисков стало возможным читать байты или записи файла в произвольном порядке или получать доступ к записям по ключу. Файлы, байты которых могут быть прочитаны в произвольном порядке, называются **файлами произвольного доступа**. Такие файлы используются многими приложениями.

Файлы произвольного доступа очень важны для многих приложений, например для баз данных. Если клиент звонит в авиакомпанию с целью зарезервировать место на конкретный рейс, программа резервирования авиабилетов должна иметь возможность получить доступ к нужной записи, не читая все тысячи предшествующих записей, содержащих информацию о других рейсах.

Для указания места начала чтения используются два метода. В первом случае каждая операция read задает позицию в файле. При втором способе используется специальная операция поиска seek, устанавливающая текущую позицию. После выполнения операции seek файл может читаться последовательно с текущей позиции.

В некоторых старых операционных системах, использовавшихся на мэйнфреймах, способ доступа к файлу (последовательный или произвольный) указывался в момент создания файла. Это позволяло операционной системе применять различные методы для хранения файлов разных классов. В современных операционных системах такого различия не проводится. Все файлы автоматически являются файлами произвольного доступа.

Атрибуты файла

У каждого файла есть имя и данные. Помимо этого все операционные системы связывают с каждым файлом также и другую информацию, например дату и время создания файла, а также его размер. Мы будем называть эти дополнительные сведения **атрибутами файла**. Список атрибутов значительно варьируется от системы к системе. В табл. 6.2 показаны некоторые возможные атрибуты, однако существуют также и другие возможности. Ни в одной существующей операционной системе не присутствуют сразу все приведенные в таблице атрибуты файлов, но каждый из них используется в той или иной системе.

Таблица 6.2. Некоторые возможные атрибуты файлов

Атрибут	Значение
Защита	Кто и каким образом может получить доступ к файлу
Пароль	Пароль для получения доступа к файлу
Создатель	Идентификатор пользователя, создавшего файл
Владелец	Текущий владелец
Флаг «только чтение»	0 — для чтения/записи; 1 — только для чтения
Флаг «скрытый»	0 — нормальный; 1 — не показывать в перечне файлов каталога
Флаг «системный»	0 — нормальный; 1 — системный
Флаг «архивный»	0 — заархивирован; 1 — требуется архивация
Флаг ASCII/двоичный	0 — ASCII; 1 — двоичный
Флаг произвольного доступа	0 — только последовательный доступ; 1 — произвольный доступ
Флаг «временный»	0 — нормальный; 1 — для удаления файла по окончании работы процесса
Флаги блокировки	0 — неблокированный; отличный от нуля для блокированного
Длина записи	Количество байтов в записи

продолжение ⇨

Таблица 6.2 (*продолжение*)

Атрибут	Значение
Позиция ключа	Смещение до ключа в записи
Длина ключа	Количество байтов в поле ключа
Время создания	Дата и время создания файла
Время последнего доступа	Дата и время последнего доступа файла
Время последнего изменения	Дата и время последнего изменения файла
Текущий размер	Количество байтов в файле
Максимальный размер	Количество байтов, до которого можно увеличивать размер файла

Первые четыре атрибута относятся к защите файла и содержат информацию о том, кто может получить доступ к файлу, а кто нет. Возможны различные схемы реализации защиты файла, несколько из них мы рассмотрим ниже. В некоторых системах пользователь должен для получения доступа к файлу указать пароль. В этом случае пароль должен входить в атрибуты файла.

Флаги представляют собой биты или короткие поля, управляющие некоторыми специфическими свойствами. Например, скрытые файлы не появляются в перечне файлов при распечатке каталога. Флаг архивации представляет собой бит, следящий за тем, была ли создана для файла резервная копия. Этот флаг очищается программой архивирования и устанавливается операционной системой при изменении файла. Таким образом программа архивирования может определить, какие файлы следует архивировать. Флаг «временный» позволяет автоматически удалять помеченный так файл по окончании работы создавшего его процесса.

Атрибуты длина записи, позиция ключа и длина ключа присутствуют только у тех файлов, записи которых могут искаться по ключу. Эти атрибуты предоставляют необходимую для поиска ключа информацию.

Различные атрибуты, хранящие значения времени, позволяют следить за тем, когда файл был создан, в последний раз изменен и когда к нему в последний раз предоставлялся доступ. Эти сведения можно использовать в различных целях. Например, если исходный файл программы был модифицирован после создания соответствующего ему объектного файла, то исходный файл должен быть перекомпилирован.

Текущий размер файла содержит количество байтов в файле в настоящий момент. В некоторых старых операционных системах, использовавшихся на мэйнфреймах, при создании файла требовалось указать также максимальную длину файла, что позволяло операционной системе зарезервировать достаточно места для последующего увеличения файла. Современные операционные системы, работающие на персональных компьютерах и рабочих станциях, умеют обходиться без подобного резервирования.

Операции с файлами

Файлы позволяют сохранять информацию и получать ее позднее. В различных операционных системах имеются различные наборы файловых операций. Ниже

мы перечислим наиболее часто встречающиеся системные вызовы, относящиеся к файлам.

1. `Create` (создание). Файл создается без данных. Этот системный вызов объявляет о появлении нового файла и позволяет установить некоторые его атрибуты.
2. `Delete` (удаление). Когда файл уже более не нужен, его удаляют, чтобы освободить пространство на диске. Этот системный вызов присутствует в каждой операционной системе.
3. `Open` (открытие). Прежде чем использовать файл, процесс должен его открыть. Системный вызов `open` позволяет системе прочитать в оперативную память атрибуты файла и список дисковых адресов для быстрого доступа к содержимому файла при последующих вызовах.
4. `Close` (закрытие). Когда все операции с файлом закончены, атрибуты и дисковые адреса более не нужны, поэтому файл следует закрыть, чтобы освободить пространство во внутренней таблице. Многие операционные системы позволяют одновременно открыть ограниченное количество файлов. Запись на диск производится поблочно, а закрытие файла вызывает запись последнего блока файла, даже если этот блок еще не заполнен до конца.
5. `Read` (чтение). Чтение данных из файла. Обычно байты поступают с текущей позиции в файле. Вызывающий процесс должен указать количество требуемых данных и предоставить для них буфер.
6. `Write` (запись). Запись данных в файл, также в текущую позицию в файле. Если текущая позиция находится в конце файла, размер файла автоматически увеличивается. В противном случае запись производится поверх существующих данных, которые теряются навсегда.
7. `Append` (добавление). Этот системный вызов представляет собой усеченную форму вызова `write`. Он может только добавлять данные к концу файла. В операционных системах с минимальным набором системных вызовов может не быть данного системного вызова.
8. `Seek` (поиск). Для файлов произвольного доступа требуется способ указать, где располагаются данные в файле. Данный системный вызов устанавливает файловый указатель в определенную позицию в файле. После выполнения данного системного вызова данные могут читаться или записываться в этой позиции.
9. `Get attributes` (получение атрибутов). Процессам часто для выполнения их работы бывает необходимо получить атрибуты файла. Например, для сборки программ, состоящих из большого числа отдельных исходных файлов, в системе UNIX часто используется программа *make*. Эта программа исследует время изменения всех исходных и объектных файлов, благодаря чему обходится компиляцией минимального количества файлов. Для выполнения этой работы ей требуется получить атрибуты файлов.

10. Set attributes (установка атрибутов). Некоторые атрибуты файла могут устанавливаться пользователем после создания файла. Этот системный вызов предоставляет такую возможность. Например, для файла может быть установлен код защиты доступа. Большинство других флагов также могут устанавливаться при помощи данного системного вызова.
11. Rename (переименование). Этот системный вызов позволяет изменить имя файла. Его присутствие в операционной системе не является необходимым, так как обычно файл можно скопировать с новым именем, а старый файл удалить.

Пример программы, использующей файловые системные вызовы

В данном разделе мы рассмотрим простую программу для операционной системы UNIX, копирующую один файл в другой. Она приведена в листинге 6.1. Эта программа обладает минимальной функциональностью и еще худшей способностью сообщать об ошибках, но тем не менее она дает достаточное представление о работе некоторых файловых системных вызовов.

Листинг 6.1. Простая программа копирования файла

```
/* Программа копирования файлов. Контроль и сообщения об ошибках минимальные. */
#include <sys/types.h>                /* включить необходимые файлы заголовков */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char * argv[]);    /* ANSI прототип */
#define BUF_SIZE 4096                 /* использовать буфер размером 4096 байт */
#define OUTPUT_MODE 0700              /* биты защиты для выходного файла */
int main(int argc, char * argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];
      if (argc != 3) exit(1);         /* если argc не равен 3, то синтаксическая ошибка */
      /* Открыть входной файл и создать выходной файл */
      in_fd = open(argv[1], O_RDONLY);        /* открыть входной файл */
      if (in_fd < 0) exit(2);                 /* если файл не может быть открыт, выход */
      out_fd = creat(argv[2], OUTPUT_MODE);   /* создать выходной файл */
      if (out_fd < 0) exit(3);                /* если файл не может быть создан, выход */
      /* Цикл копирования */
      while (TRUE) {
         rd_count = read(in_fd, buffer, BUF_SIZE);      /* прочитать блок данных */
         if (rd_count <= 0) break;                      /* если конец файла или ошибка,
                                                        /* выйти из цикла */
         wt_count = write(out_fd, buffer, rd_count);    /* записать данные */
         if (wt_count <= 0) exit(4);                    /* wt_count <= 0 является ошибкой */
      }
      /* Закрыть оба файла */
      close(in_fd);
      close(out_fd);
      if (rd_count == 0)                   /* при последнем чтении ошибки не было */
         exit(0);
      else
         exit(5);                          /* ошибка при последнем чтении */
}
```

Программа *copyfile* может вызываться, например, при помощи командной строки

```
copyfile abc xyz
```

чтобы скопировать файл *abc* в файл *xyz*. Если файл *xyz* уже существует, он будет перезаписан. В противном случае он будет создан. Программа должна вызываться обязательно с двумя аргументами, каждый из которых представляет собой допустимое имя файла.

Четыре оператора *#include* в начале программы обеспечивают включение в программу большого количества определений и прототипов функций. Это требуется для того, чтобы программа удовлетворяла международным стандартам, но не будет интересовать нас в дальнейшем. Следующая строка содержит прототип функции *main*, что требуется стандартом ANSI C, но также не интересует нас в настоящий момент.

Первый оператор *#define* является макроопределением, определяющим строку *BUF_SIZE* как макрос, заменяющийся в тексте при компиляции числом 4096. Программа будет читать и писать данные кусками по 4096 байт. Создавать подобные константы и использовать их вместо указания напрямую чисел в программе считается хорошим стилем программирования. При этом программа не только становится более удобочитаемой, но ее также проще и изменять в случае необходимости. Второй оператор *#define* определяет круг пользователей, способных получить доступ к выходному файлу.

Основная программа называется *main*. У нее есть два аргумента, *argc* и *argv*. Этим аргументам присваивается значение операционной системой при вызове программы. Первый аргумент сообщает количество параметров (слов) в командной строке, включая имя программы. Он должен быть равен трем. Второй аргумент представляет собой массив указателей на текстовые строки, содержащие параметры командной строки. В данном примере элементы этого массива будут содержать указатели на следующие строки:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

Именно из этого массива программа может получить входные параметры.

В программе объявляются пять переменных. Первые две переменные, *in_fd* и *out_fd*, предназначаются для хранения **дескрипторов файлов**, представляющих собой небольшие целые числа, возвращаемые процедурами открытия или создания файла. Следующие две переменные, *rd_count* и *wt_count*, являются байтовыми счетчиками, возвращаемыми, соответственно, процедурами `read` и `write`. Последняя переменная, *buffer*, представляет собой символьный массив, используемый в качестве буфера для чтения и записи данных.

Первый исполняемый оператор проверяет равенство счетчика аргументов *argc* трем. Если счетчик *argc* не равен трем, программа завершается с кодом 1. Любой код завершения программы, отличный от 0, означает ошибку. Код завершения программы является единственным способом сообщения об ошибках, применяемым в данной программе.

Затем программа пытается открыть входной файл и создать выходной файл. Если открытие файла проходит успешно, операционная система присваивает

переменной *in_fd* положительное значение, соответствующее дескриптору открытого файла. При последующих операциях с файлом это число должно указывать операционной системе, к какому файлу относится данный системный вызов. Соответственно, если удается успешно создать выходной файл, переменной *out_fd* присваивается значение дескриптора файла, использующегося для его идентификации. Второй аргумент процедуры *creat* устанавливает код защиты создаваемого файла. Если одна из этих операций завершается неудачей, то вместо дескриптора файла возвращается значение –1 и программа выходит с соответствующим кодом ошибки.

Затем начинается цикл копирования с попытки чтения в буфер *buffer* 4 Кбайт данных. Чтение выполняется при помощи библиотечной процедуры *read*, обращающейся к системному вызову read. Первый параметр идентифицирует файл, второй указывает на буфер, а третий сообщает, сколько байтов следует прочитать. Переменной *rd_count* присваивается значение, равное количеству прочитанных байтов. В нормальной ситуации это будет 4096 или меньшее число, равное величине оставшейся части файла. При достижении конца файла переменной *rd_count* будет присвоено число 0. Когда значение переменной *rd_count* станет отрицательным или нулевым, это будет означать, что операцию копирования продолжать более невозможно и цикл чтения и записи будет прерван оператором *break*.

Обращение к библиотечной процедуре *write* записывает содержимое буфера в выходной файл. Первый параметр указывает файл, второй — буфер, а третий содержит количество байтов, которые необходимо записать, подобно библиотечной процедуре *read*. Обратите внимание, что записывается именно столько байтов, сколько было прочитано, а не весь буфер *BUF_SIZE*. Этот момент важен, так как последняя операция чтения прочитает не 4096 байт, если только длина файла не кратна 4 Кбайт.

Когда весь файл будет прочитан, первое обращение к библиотечной процедуре *read* вернет значение 0, которое будет присвоено переменной *rd_count*, и цикл прервется. После этого оба файла закрываются, и программа прекращает свою работу с нулевым статусом, соответствующим нормальному завершению.

Хотя системные вызовы Windows отличаются от системных вызовов UNIX, общая структура программы копирования файлов, работающей в режиме командной строки в Windows, схожа с программой, приведенной в листинге 6.1. Мы обсудим системные вызовы Windows 2000 в главе 11.

Файлы, отображаемые на адресное пространство памяти

Многие программисты считают описанный в предыдущем примере доступ неудобным, особенно по сравнению с доступом к обычной памяти. По этой причине в некоторых операционных системах, начиная с системы MULTICS, был предоставлен способ отображения файлов на адресное пространство работающего процесса. Концептуально можно представить себе два новых системного вызова, map и unmap. Первый системный вызов принимает на входе два параметра: имя файла и виртуальный адрес памяти, по которому операционная система отображает указанный файл.

Предположим, например, что некий файл размером 64 Кбайт отображается на виртуальную память начиная с адреса 512 К. После этого любая команда процессора, читающая байт по адресу 512 К, получит байт 0 этого файла и т. д. Запись по адресу 512 К+21000 изменит байт 21000 файла. Когда процесс завершает свою работу, модифицированный файл остается на диске, как если бы он был изменен системными вызовами seek и write.

Для реализации отображения файлов на память изменяются системные внутренние таблицы. При обращении к памяти по адресу от 512 до 576 К происходит прерывание из-за отсутствия страницы, обработчик которого предоставляет считанную в память страницу 0 файла. При записи происходит приблизительно то же самое, но страница памяти, на которую отображается страница файла, помечается как модифицированная. Если потом эта страница удаляется из памяти алгоритмом замены страниц, она записывается в соответствующее место файла. После завершения процесса все модифицированные страницы сохраняются в соответствующих файлах.

Отображение файлов на память лучше всего работает в операционной системе, поддерживающей сегментацию. В такой системе каждый файл может быть отображен на свой собственный сегмент, так чтобы байт *k* файла был также байтом *k* сегмента. На рис. 6.3, *а* показан процесс с двумя сегментами, исполняемым кодом программы и данными. Предположим, что процесс копирует файлы подобно программе из листинга 6.1. Сначала он отображает на сегмент исходный файл, например *abc*. Затем он создает пустой сегмент и отображает его на выходной файл, *xyz*. В результате получается ситуация, показанная на рис. 6.3, *б*.



Рис. 6.3. Сегментированный процесс до отображения файлов на адресное пространство (а); процесс после отображения существующего файла *abc* на один сегмент и создания нового сегмента для файла *xyz* (*б*)

В этот момент процесс может скопировать сегмент-источник в сегмент-приемник с помощью обычного цикла копирования памяти. При этом не требуются системные вызовы read и write. Когда копирование закончено, процесс может выполнить системный вызов unmap, чтобы удалить эти файлы из адресного пространства, и завершить свою работу. Выходной файл, *xyz*, теперь будет существовать на диске, как если бы он был создан обычным путем.

Хотя отображение на память устраняет необходимость обращения к системным вызовам ввода-вывода, вместе с ним появляются новые проблемы. Во-первых, в нашем примере операционной системе трудно определить длину выходного

файла *xyz*. Операционная система знает номер максимальной модифицированной страницы, но она не может определить, сколько байтов было записано в эту страницу. Предположим, программа использует только страницу 0 и после выполнения цикла копирования все байты остались равны 0 (их исходному значению). Возможно, файл *xyz* состоит из 10 нулей. Может быть, он должен состоять из 100 нулей. Кто знает? Операционная система не может этого сказать. Все, что она может сделать — это создать файл, длина которого равна размеру страницы.

Вторая проблема может возникнуть при попытке одного процесса открыть файл, уже отображенный на адресное пространство другого процесса. Если первый процесс модифицирует страницу, это изменение не отразится на файле до тех пор, пока эта страница не будет сохранена в файле. Операционная система должна прилагать особые усилия, чтобы гарантировать, что оба процесса не работают с устаревшими версиями файла.

Третья проблема, связанная с отображением файлов на память, вызвана тем, что файл может оказаться больше сегмента памяти и даже больше чем все виртуальное адресное пространство. При этом единственный способ работы системного вызова `map` состоит в отображении на память части файла. Хотя такой метод и работает, он все же менее удобен, чем отображение всего файла целиком.

Каталоги

В файловых системах файлы обычно организуются в **каталоги** или **папки**, которые, в свою очередь, в большинстве операционных систем также являются файлами. В данном разделе мы рассмотрим каталоги, их организацию, свойства и действия, которые могут быть выполнены с ними.

Одноуровневые каталоговые системы

Простейшая форма системы каталогов состоит в том, что имеется один каталог, в котором содержатся все файлы. Иногда его называют **корневым каталогом**, но поскольку он в таких системах единственный, его название не имеет значение. Такая система была весьма распространена на ранних персональных компьютерах, в частности потому, что у них было всего по одному пользователю. Первый в мире суперкомпьютер CDC 6600[1] также имел всего один каталог для всех файлов, несмотря на то, что на нем одновременно работало много пользователей. Это решение было принято для сохранения простоты программного обеспечения.

Схематично однокаталоговая система показана на рис. 6.4. В данном примере каталог состоит из четырех файлов. На рисунке буквами *A*, *B* и *C* показаны не имена файлов, а их *владельцы* (так как именно наличие нескольких пользователей в такой системе создает проблемы). Преимуществом такой схемы является ее простота и способность быстро находить файлы, так как они могут располагаться только в одном месте.

---

[1] Разработан Сэймуром Крэйем в корпорации Control Data Corp. в 1964 году. — *Примеч. перев.*

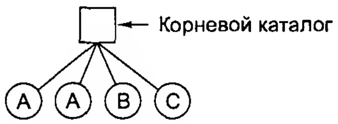


Рис. 6.4. Однокаталоговая система, содержащая четыре файла

Недостаток системы с одним каталогом и несколькими пользователями состоит в том, что различные пользователи могут случайно использовать для своих файлов одинаковые имена. Например, если пользователь *A* создаст файл *mailbox*, а затем пользователь *B* также создаст файл *mailbox*, то файл, созданный пользователем *B*, запишется поверх файла, созданного пользователем *A*. Поэтому такая схема более не используется в многопользовательских системах, но может применяться в небольших встроенных системах, например автомобильной системе, предназначенной для хранения профилей пользователей для небольшого количества водителей.

Двухуровневая система каталогов

Первым этапом в деле решения проблемы одинаковых имен файлов, созданных различными пользователями, можно считать систему, в которой каждому пользователю выделяется один каталог. При этом имена файлов, созданных одним пользователем, не конфликтуют с именами файлов другого пользователя. Схематично такая двухуровневая каталоговая система проиллюстрирована на рис. 6.5. Буквы обозначают владельцев каталогов и файлов. Такая организация могла, например, использоваться на многопользовательском компьютере или в простой сети персональных компьютеров, соединенных с общим файловым сервером локальной сетью.

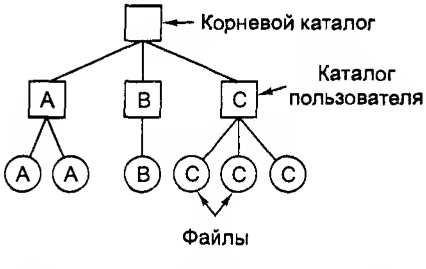


Рис. 6.5. Двухуровневая каталоговая система

Когда при такой схеме пользователь пытается открыть файл, система знает, что это за пользователь, и ищет файл в соответствующем каталоге. Следовательно, для работы в такой системе требуется начальная регистрация пользователя, при которой пользователь указывает свое имя или идентификатор. В одноуровневой каталоговой системе такая процедура не требовалась.

При реализации такой системы в ее базовой форме пользователи могут получать доступ только к файлам в своем собственном каталоге. Однако небольшая модификация основной схемы позволяет пользователям получать доступ к фай-

лам других пользователей. Для этого им нужно указать идентификатор владельца файла. Например, команда

```
open("x")
```

может быть вызовом для открытия файла *x* в каталоге пользователя, а команда

```
open("nancy/x")
```

может быть вызовом для открытия файла *x* в каталоге другого пользователя, Нэнси.

Одна из ситуаций, в которой пользователям может понадобиться получить доступ к файлам, не находящимся в их каталогах, — это выполнение системных двоичных программ. Копирование всех системных программ во все пользовательские каталоги крайне неэффективно. Таким образом, возникает необходимость в создании по крайней мере одного системного каталога, содержащего все исполнимые двоичные системные файлы.

Иерархические каталоговые системы

Благодаря двухуровневой иерархии исчезают конфликты имен файлов между различными пользователями, но ее недостаточно для пользователей с большим числом файлов. Обычно пользователям бывает необходимо логически группировать свои файлы. Например, у профессора может быть набор файлов, образующих книгу, которую он пишет для одного курса, другое множество файлов, содержащее программы студентов для иного курса. Третий набор файлов может содержать исходные тексты разрабатываемого им нового компилятора, четвертая группа файлов — предложения различных грантов, а также электронную почту, расписание собраний, статьи, игры и т. д. Требуется некий гибкий способ, позволяющий объединять эти файлы в группы.

Следовательно, нужна некая общая иерархия (то есть дерево каталогов). При таком подходе каждый пользователь может сам создать себе столько каталогов, сколько ему нужно, группируя свои файлы естественным образом. Этот подход проиллюстрирован на рис. 6.6. Здесь каталоги *A*, *B* и *C*, содержащиеся в корневом каталоге, принадлежат различным пользователям, два из которых создали подкаталоги для проектов, над которыми они работают.

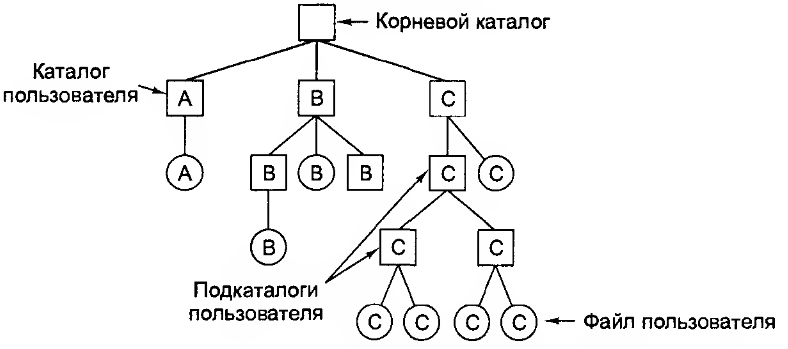


Рис. 6.6. Иерархическая каталоговая система

Возможность создавать произвольное количество подкаталогов является мощным структурирующим инструментом, позволяющим пользователям организовать свою работу. По этой причине почти все современные файловые системы организованы подобным образом.

Имя пути

При организации файловой системы в виде дерева каталогов требуется некоторый способ указания файла. Для этого обычно используются два различных метода. В первом случае каждому файлу дается **абсолютное имя пути**, состоящее из имен всех каталогов от корневого до того, в котором содержится файл, и имени самого файла. Например, путь */usr/ast/mailbox* означает, что корневой каталог содержит подкаталог *usr*, который, в свою очередь, содержит подкаталог *ast*, где находится файл *mailbox*. Абсолютные имена путей всегда начинаются от корневого каталога и являются уникальными. В системе UNIX компоненты пути разделяются косой чертой /. В Windows в качестве разделителя используется обратная косая черта \. В системе MULTICS использовался символ >. Таким образом, одно и то же имя пути в этих трех операционных системах будет выглядеть следующим образом:

```
Windows    \usr\ast\mailbox
UNIX       /usr/ast/mailbox
MULTICS    >usr>ast>mailbox
```

Если первой буквой имени пути был разделитель, это означало, независимо от используемого в качестве разделителя символа, что путь абсолютный.

Применяется и **относительное имя пути**. Оно используется вместе с концепцией **рабочего каталога** (также называемого **текущим каталогом**). Пользователь может назначить один из каталогов текущим рабочим каталогом. В этом случае все имена путей, не начинающиеся с символа разделителя, считаются относительными и отсчитываются относительно текущего каталога. Например, если текущим каталогом является */usr/ast*, тогда к файлу с абсолютным путем */usr/ast/mailbox* можно обратиться просто как к *mailbox*. Другими словами, команда UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

и команда

```
cp mailbox mailbox.bak
```

выполнят одно и то же действие, если рабочим каталогом является */usr/ast*. Относительная форма часто оказывается более удобной, но она выполняет то же самое, что и абсолютная.

Некоторым программам бывает нужно получить доступ к файлам независимо от того, какой каталог является в данный момент текущим. В этом случае они всегда должны использовать абсолютные имена. Например, программе проверки правописания может понадобиться для выполнения работы прочитать файл */usr/lib/dictionary*. В этом случае она должна использовать полное, абсолютное имя файла, так как она не знает, каким будет рабочий каталог при ее вызове. Абсолютное имя файла будет работать всегда, независимо от того, какой каталог является текущим в данный момент.

Если программе проверки правописания понадобится большое количество файлов из каталога */usr/lib*, она может, обратившись к операционной системе, поменять рабочий каталог на */usr/lib*, после чего использовать просто имя *dictionary* для первого параметра системного вызова open. Явно указав свой рабочий каталог, программа может использовать в дальнейшем относительные имена, так как точно знает, где она находится в дереве каталогов.

У каждого процесса есть свой рабочий каталог, поэтому, когда процесс меняет свой рабочий каталог и потом завершает работу, это не влияет на работу других процессов, и в файловой системе не остается никаких следов от подобных изменений рабочих каталогов. Таким образом, процесс может спокойно менять свой рабочий каталог, когда это ему удобно. С другой стороны, если *библиотечная процедура* поменяет свой рабочий каталог и не восстановит его при возврате управления, программа, вызвавшая ее, может оказаться не в состоянии продолжать свою работу, так как ее предположения о текущем каталоге окажутся неверными. По этой причине библиотечные процедуры редко меняют свои рабочие каталоги, а когда все-таки меняют, то обязательно восстанавливают рабочий каталог перед возвратом.

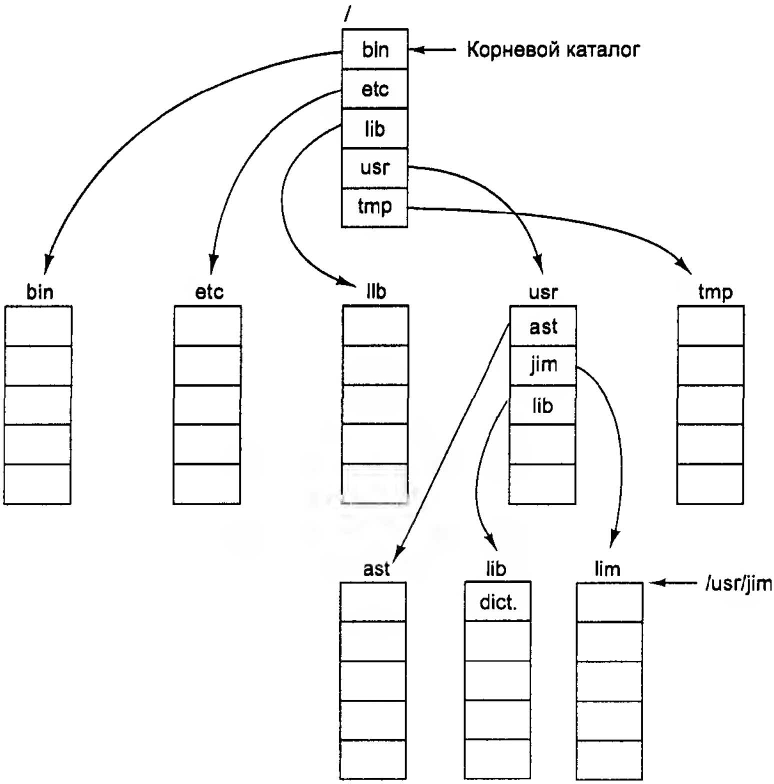


Рис. 6.7. Дерево каталогов UNIX

Большинство операционных систем, поддерживающих иерархические каталоги, имеют специальные элементы в каждом каталоге. Это «.» и «..», означающие текущий каталог и родительский каталог. Чтобы продемонстрировать, как это

работает, обратимся к дереву каталогов системы UNIX, показанному на рис. 6.7. Для некоторого процесса каталог */usr/ast* является рабочим. Чтобы переместиться вверх по дереву, он может использовать обозначение «..». Например, он может копировать файл */usr/lib/dictionary* в свой собственный каталог при помощи команды

```
cp ../lib/dictionary .
```

Две точки являются инструкцией системе подняться вверх (в каталог *usr*). После этого нужно открыть каталог *lib* и найти в нем файл *dictionary*.

Одиночная точка означает текущий каталог. Когда команда *cp* в качестве второго аргумента получает точку, она интерпретирует ее как текущий каталог и копирует все файлы туда. Конечно, ту же команду можно было задать и так:

```
cp /usr/lib/dictionary .
```

Здесь использование точки позволяет сэкономить время, затрачиваемое пользователем на набор слова *dictionary* второй раз. Тем не менее команда

```
cp /usr/lib/dictionary dictionary
```

также прекрасно работает и делает то же самое, что и команда

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Все эти команды выполняют одни и те же действия.

Операции с каталогами

Системные вызовы, управляющие каталогами, значительно менее схожи в различных системах, чем системные вызовы для работы с файлами. Чтобы дать представление о том, что они собой представляют и как работают, приведем следующий пример (взятый из UNIX).

1. `Create`. Создать каталог. Только что созданный каталог пуст и не содержит других элементов, кроме «.» и «..», автоматически помещаемых в каталог операционной системой[1].
2. `Delete`. Удалить каталог. Может быть удален только пустой каталог. Элементы «.» и «..» файлами не являются и удалены быть не могут.
3. `Opendir`. Открыть каталог. После этой операции каталог может быть прочитан. Например, для распечатки всех файлов, содержащихся в каталоге, программа, создающая листинг, открывает каталог, чтобы прочитать имена всех содержащихся в нем файлов. Прежде чем каталог может быть прочитан, его следует открыть, подобно открытию и чтению файла.
4. `Closedir`. Закрыть каталог. Когда каталог прочитан, его следует закрыть, чтобы освободить место во внутренней таблице.
5. `Readdir`. Прочитать следующий элемент открытого каталога. В прежние времена было возможно читать каталоги с помощью обычного системного вызова `read`, но такой подход был небезопасен, так как требовал от программиста

[1] Точнее, это не элементы, а обозначения, принятые в командной строке и отображаемые в листингах, которым не обязательно соответствуют отдельные элементы каталогов. — *Примеч. перев.*

умения работать с внутренней структурой каталогов. Поэтому был создан отдельный системный вызов readdir, всегда возвращающий одну запись каталога стандартного формата независимо от используемой структуры каталогов.

6. Rename. Переименование каталога. Во многих отношениях каталоги аналогичны файлам и могут переименовываться так же, как и файлы.
7. Link. Связывание представляет собой технику, позволяющую файлу появляться сразу в нескольких каталогах. Этот системный вызов принимает в качестве входных параметров имя файла и имя пути и создает связь между ними. Таким образом, один и тот же файл может появляться сразу в нескольких каталогах. Подобная связь, увеличивающая на единицу счетчик i-узла файла (для учета количества каталогов со ссылками на этот файл), иногда называется **жесткой связью**.
8. Unlink. Удаление ссылки на файл из каталога. Если файл присутствует только в одном каталоге, то данный системный вызов удалит его из файловой системы. Если существует несколько ссылок на этот файл, то будет удалена только указанная ссылка, а остальные останутся. Этот системный вызов применяется для удаления файла в операционной системе UNIX.

Приведенный выше список содержит наиболее важные системные вызовы, но существует также множество других, например для управления защитой информации.

Реализация файловой системы

Теперь пора перейти от рассмотрения файловой системы с точки зрения пользователя к рассмотрению с точки зрения разработчика системы. Пользователей интересует то, как называются файлы, какие операции возможны с ними, как выглядит дерево каталогов и тому подобные интерфейсные вопросы. Проектировщики файловых систем интересуются тем, как хранятся файлы и каталоги, как осуществляется управление дисковым пространством и как добиться надежной и эффективной работы файловой системы. В следующих разделах мы познакомимся с этой точкой зрения на файловую систему.

Структура файловой системы

Файловые системы хранятся на дисках. Большинство дисков делятся на несколько разделов с независимой файловой системой на каждом разделе. Сектор 0 диска называется **главной загрузочной записью** (MBR, Master Boot Record) и используется для загрузки компьютера. В конце главной загрузочной записи содержится таблица разделов. В этой таблице хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из разделов помечен в таблице как активный. При загрузке компьютера BIOS считывает и исполняет MBR-запись, после чего загрузчик в MBR-записи определяет активный раздел диска, считывает его первый блок, называемый **загрузочным**, и исполняет его. Программа, находящаяся в загрузочном блоке, загружает операционную систему, содержащуюся в этом разделе.

Для единообразия каждый дисковый раздел начинается с загрузочного блока, даже если в нем не содержится загружаемой операционной системы. К тому же в этом разделе может быть в дальнейшем установлена операционная система, поэтому зарезервированный загрузочный блок оказывается полезным.

Во всем остальном строение раздела диска меняется от системы к системе. Часто файловые системы содержат некоторые из элементов, показанных на рис. 6.8. Один из таких элементов, называемый **суперблоком**, содержит ключевые параметры файловой системы и считывается в память при загрузке компьютера или при первом обращении к файловой системе. Типичная информация, хранящаяся в суперблоке, включает «магическое» число, позволяющее различать системные файлы, количество блоков в файловой системе, а также другую ключевую административную информацию.

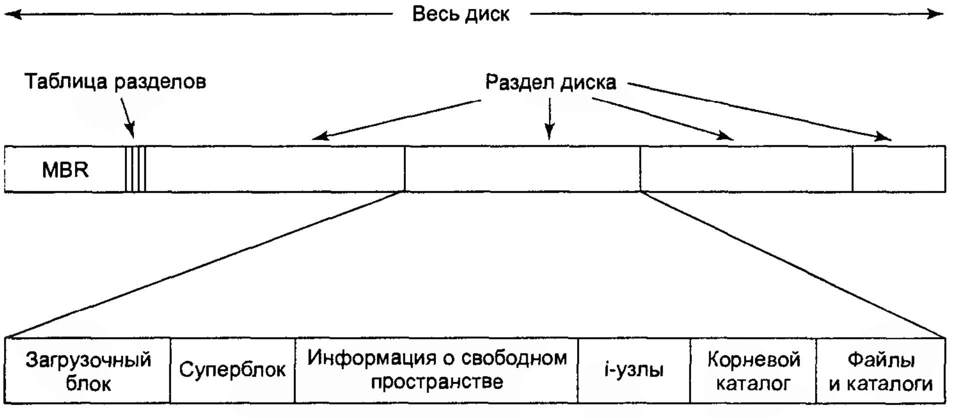


Рис. 6.8. Возможная структура файловой системы

Следом располагается информация о свободных блоках файловой системы, например в виде битового массива или списка указателей. За этими данными может следовать информация об i-узлах, представляющих собой массив структур данных, по одной структуре на файл, содержащих всю информацию о файлах. Следом может размещаться корневой каталог, содержащий вершину дерева файловой системы. Наконец, остальное место дискового раздела занимают все остальные каталоги и файлы.

Реализация файлов

Вероятно, наиболее важным моментом в реализации хранения файлов является учет соответствия блоков диска файлам. Для определения того, какой блок какому файлу принадлежит, в различных операционных системах применяются различные методы. Некоторые из них будут рассмотрены в данном разделе.

Непрерывные файлы

Простейшей схемой выделения файлам определенных блоков на диске является система, в которой файлы представляют собой непрерывные наборы соседних блоков диска. Тогда на диске, состоящем из блоков по 1 Кбайт, файл размером

в 50 Кбайт будет занимать 50 последовательных блоков. При 2-килобайтных блоках такой файл займет 25 соседних блоков.

Пример непрерывных файлов показан на рис. 6.9, *а*. Здесь показаны первые 40 блоков диска, начиная с блока 0, слева. Вначале диск был пуст. Затем на диск, начиная с блока 0, был записан файл *A* длиной в четыре блока. После него был записан шестиблочный файл *B*, впритык к файлу *A*. Обратите внимание, что каждый файл начинается с нового блока, так что если длина файла *A* была равна 3½ блока, некоторое место в конце последнего блока файла пропадает. На рисунке всего показано семь файлов. Каждый следующий файл начинается с блока, следующего за последним блоком предыдущего файла. Затенение используется только для того, чтобы было легче различать отдельные файлы.

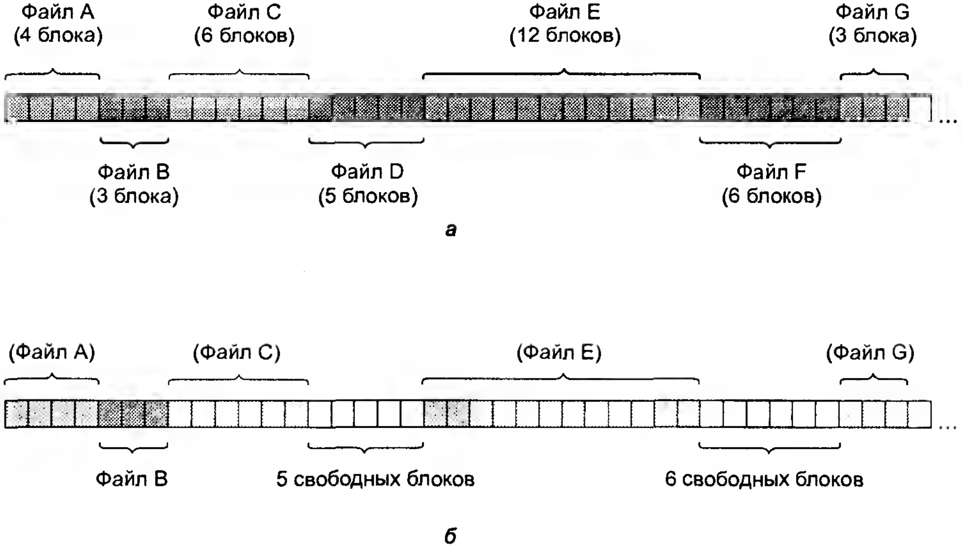


Рис. 6.9. Семь непрерывных файлов на диске (*а*); состояние диска после удаления двух файлов (*б*)

У непрерывных файлов есть два существенных преимущества. Во-первых, такую систему легко реализовать, так как системе, чтобы определить, какие блоки принадлежат тому или иному файлу, нужно следить всего лишь за двумя числами: номером первого блока файла и числом блоков в файле. Зная первый блок файла, любой другой его блок легко получить при помощи простой операции сложения.

Во-вторых, при работе с непрерывными файлами производительность просто превосходна, так как весь файл может быть прочитан с диска за одну операцию. Требуется только одна операция поиска (для первого блока). После этого более не нужно искать цилиндры и тратить время на ожидания вращения диска, поэтому данные могут считываться с максимальной скоростью, на которую способен диск. Таким образом, непрерывные файлы легко реализуются и обладают высокой производительностью.

К сожалению, у такого способа распределения дискового пространства имеется серьезный недостаток: со временем диск становится фрагментированным. Чтобы

понять, как это происходит, рассмотрим рис. 6.9, *б*. Два файла, *D* и *F*, были удалены. Когда файл удаляется, его блоки освобождаются, оставляя промежутки свободных блоков на диске. По мере удаления файлов диск становится все более «дырявым».

Вначале эта фрагментация не представляет проблемы, так как каждый новый файл может быть записан в конец диска, вслед за предыдущим файлом. Однако в конце концов диск заполнится и либо потребуется специальная операция по уплотнению используемого пространства диска, либо надо будет изыскать способ использовать свободное пространство на месте удаленных файлов. Для повторного использования освободившегося пространства потребуется содержать список пустых участков, что в принципе выполнимо. Однако при создании нового файла будет необходимо знать его окончательный размер, чтобы выбрать для него участок подходящего размера.

Представьте себе последствия такой структуры. Пользователь запускает текстовый редактор или текстовой процессор, чтобы создать документ. Первое, что интересует программу, это сколько байтов будет в документе. На этот вопрос следует дать ответ, в противном случае программа не сможет работать. Если пользователь укажет слишком маленькое число, программа может закончиться аварийно, так как свободный участок диска окажется заполнен и будет негде разместить остальную часть файла. Если пользователь попытается обойти эту проблему, задав заведомо большой окончательный размер, например 100 Мбайт, может случиться, что редактор не сможет найти такой большой свободный участок и сообщит, что не может создать файл. Конечно, пользователь может поторговаться и снизить свои требования до 50 Мбайт и т. д. до тех пор, пока не найдется подходящий свободный участок. Тем не менее такая схема вряд ли доставит удовольствие пользователям.

И все-таки есть ситуации, в которых непрерывные файлы могут применяться и в самом деле широко используются: на компакт-дисках. Здесь все размеры файлов известны заранее и не могут меняться при последующем использовании файловой системы CD-ROM. Наиболее распространенную файловую систему CD-ROM мы рассмотрим ниже в этой главе.

Как уже говорилось в главе 1, в кибернетике история часто повторяется с появлением новых технологий. Файловые системы, состоящие из непрерывных файлов, применялись на магнитных дисках много лет назад благодаря их простоте и высокой производительности (удобство для пользователей почти не принималось тогда в расчет). Затем эта идея была позабыта из-за необходимости задавать окончательный размер файла при его создании. Но с появлением CD-ROM и DVD, а также других одноразовых оптических носителей о преимуществах непрерывных файлов вспомнили снова. Изучение старых систем и идей оказывается полезным, так как многие простые и ясные концепции тех систем находят применение в новых системах самым удивительным образом.

Связные списки

Второй метод размещения файлов состоит в представлении каждого файла в виде связного списка из блоков диска, как показано на рис. 6.10. Первое слово каждого блока используется как указатель на следующий блок. В остальной части блока хранятся данные.

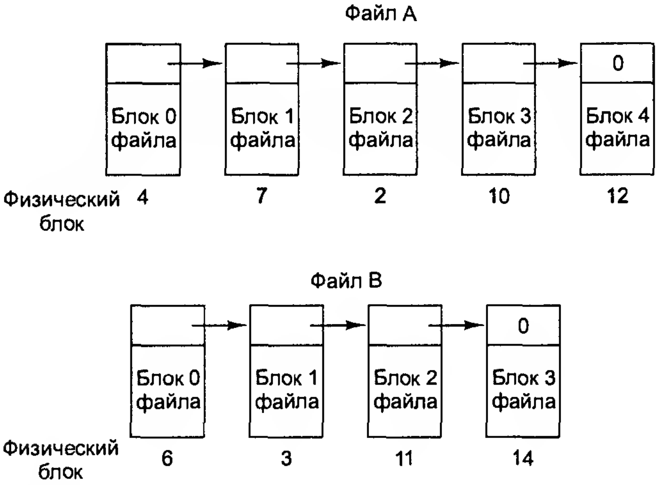


Рис. 6.10. Размещение файла в виде связного списка блоков диска

В отличие от систем с непрерывными файлами, такой метод позволяет использовать каждый блок диска. Нет потерь дискового пространства на фрагментацию (кроме потерь в последних блоках файла). Кроме того, в каталоге нужно хранить только адрес первого блока файла. Всю остальную информацию можно найти там.

С другой стороны, хотя последовательный доступ к такому файлу несложен, произвольный доступ будет довольно медленным. Чтобы получить доступ к блоку n, операционная система должна сначала прочитать первые $n - 1$ блоков по очереди. Очевидно, такая схема оказывается очень медленной.

Кроме того, размер блока уменьшается на несколько байтов, требуемых для хранения указателя. Хотя это и не смертельно, но размер блока, не являющийся степенью двух, будет менее эффективным, так как многие программы читают и пишут блоками по 512, 1024, 2048 и т. д. байтов. Если первые несколько байтов каждого блока будут заняты указателем на следующий блок, то для чтения блока полного размера придется считывать и объединять два соседних блока диска, для чего потребуется выполнение дополнительных операций.

Связный список при помощи таблицы в памяти

Оба недостатка предыдущей схемы организации файлов в виде связных списков могут быть устранены, если указатели на следующие блоки хранить не прямо в блоках, а в отдельной таблице, загружаемой в память. На рис. 6.11 показан внешний вид такой таблицы для файлов с рис. 6.10. На обоих рисунках показаны два файла. Файл *A* использует блоки диска *4*, *7*, *2*, *10* и *12*, а файл *B* использует блоки диска *6*, *3*, *11* и *14*. С помощью таблицы, показанной на рис. 6.11, мы можем начать с блока *4* и следовать по цепочке до конца файла. То же может быть сделано для второго файла, если начать с блока *6*. Обе цепочки завершаются специальным маркером (например –1), не являющимся допустимым номером блока. Такая таблица, загружаемая в оперативную память, называется **FAT-таблицей** (File Allocation Table — таблица размещения файлов).



Рис. 6.11. Таблица размещения файлов

Эта схема позволяет использовать для данных весь блок. Кроме того, случайный доступ при этом становится намного проще. Хотя для получения доступа к какому-либо блоку файла все равно понадобится проследовать по цепочке по всем ссылкам вплоть до ссылки на требуемый блок, однако в данном случае вся цепочка ссылок уже хранится в памяти, поэтому для следования по ней не требуются дополнительные дисковые операции. Как и в предыдущем случае, в каталоге достаточно хранить одно целое число (номер начального блока файла) для обеспечения доступа ко всему файлу.

Основной недостаток этого метода состоит в том, что вся таблица должна постоянно находиться в памяти. Для 20-гигабайтного диска с блоками размером 1 Кбайт потребовалась бы таблица из 20 млн записей, по одной для каждого из 20 млн блоков диска. Каждая запись должна состоять как минимум из трех байтов[1]. Для ускорения поиска размер записей должен быть увеличен до 4 байт. Таким образом, таблица будет постоянно занимать 60 или 80 Мбайт оперативной памяти. Таблица, конечно, может быть размещена в виртуальной памяти, но и в этом случае ее размер оказывается чрезмерно большим, к тому же постоянная выгрузка таблицы на диск и загрузка с диска существенно снизит производительность файловых операций.

I-узлы

Последний метод отслеживания принадлежности блоков диска файлам состоит в связывании с каждым файлом структуры данных, называемой **i-узлом** (index node — индекс-узел), содержащей атрибуты файла и адреса блоков файла. Простой

[1] Трех байтов будет недостаточно, так как 2^{24} = 16 777 216, что чуть меньше 20 млн. — *Примеч. перев.*

пример i-узла показан на рис. 6.12. При наличии i-узла можно найти все блоки файла. Большое преимущество такой схемы перед хранящейся в памяти таблицей из связных списков заключается в том, что каждый конкретный i-узел должен находиться в памяти только тогда, когда соответствующий ему файл открыт. Если каждый i-узел занимает n байт, а одновременно открыто может быть k файлов, то для массива i-узлов потребуется в памяти всего kn байтов.

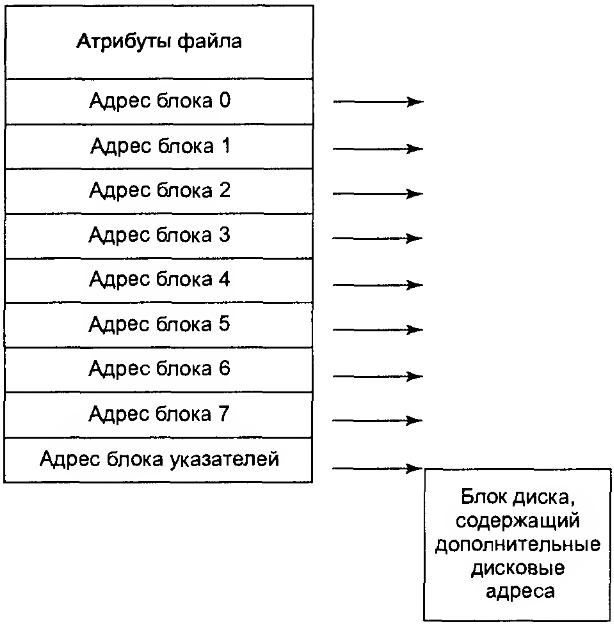


Рис. 6.12. Пример i-узла

Обычно этот размер значительно меньше, чем FAT-таблица, описанная в предыдущем разделе. Это легко объясняется. Размер таблицы, хранящей связный список всех блоков диска, пропорционален размеру самого диска. Для диска из n блоков потребуется n записей в таблице. Таким образом, размер таблицы линейно[1] растет с ростом размера диска. Для схемы i-узлов, напротив, требуется массив в памяти с размером, пропорциональным максимальному количеству файлов, которые могут быть открыты одновременно. При этом не важно, будет ли размер диска 1 Гбайт, 10 Гбайт или 100 Гбайт.

С такой схемой связана проблема, заключающаяся в том, что при выделении каждому файлу фиксированного количества дисковых адресов этого количества может не хватить. Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для следующего адресного блока, как показано на рис. 6.12. Более того, можно создавать целые цепочки и даже деревья адресных блоков. Мы снова вернемся к теме i-узлов, когда приступим к изучению системы UNIX позднее.

---

[1] Даже быстрее, чем линейно, так как с увеличением числа блоков требуется все большее число битов для хранения их номеров. — *Примеч. перев.*

Реализация каталогов

Прежде чем прочитать файл, его следует открыть. При открытии файла операционная система использует поставляемое пользователем имя пути, чтобы найти запись в каталоге. Запись в каталоге содержит информацию, необходимую для нахождения блоков диска. В зависимости от системы это может быть дисковый адрес всего файла (для непрерывных файлов), номер первого блока файла (обе схемы связных списков) или номер i-узла. Во всех случаях основная функция каталоговой системы состоит в преобразовании ASCII-имени в информацию, необходимую для нахождения данных.

С этой проблемой тесно связан вопрос размещения атрибутов файла. Каждая файловая система поддерживает различные атрибуты файла, такие как дату создания файла, имя владельца файла и т. д., и всю эту информацию нужно где-то хранить. Один из возможных вариантов состоит в хранении этих сведений прямо в каталоговой записи. Многие файловые системы именно так и поступают. Этот вариант показан на рис. 6.13, *а*. В этой простой схеме каталог состоит из списка элементов фиксированной длины по одному на файл, содержащих имена файлов, структуру атрибутов файла, а также один или несколько дисковых адресов, указывающих расположение файла на диске.

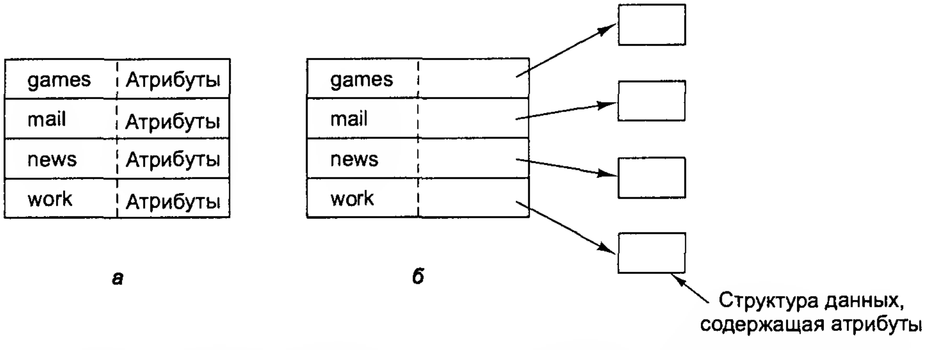


Рис. 6.13. Простой каталог, содержащий записи фиксированной длины с атрибутами и дисковыми адресами (*а*); каталог, в котором каждая запись является просто ссылкой на i-узел (*б*)

Системы, использующие i-узлы, могут хранить атрибуты в i-узлах, а не в записях каталога. В этом случае запись в каталоге может быть короче: просто имя файла и номер i-узла. Этот подход показан на рис. 6.13, *б*. Как мы увидим позднее, у этого метода есть определенные преимущества по сравнению с помещением атрибутов прямо в каталоговые записи. Два подхода, показанные на рис. 6.13, соответствуют системам MS-DOS и UNIX, о чем еще будет рассказано в этой главе.

До сих пор мы предполагали, что файлы имеют короткие имена фиксированной длины. В системе MS-DOS файл может иметь имя длиной от 1 до 8 символов, а также расширение длиной до 3 символов. В системе UNIX Version 7 имена файлов могут быть от одного до 14 символов, включая расширение. Однако почти всеми современными операционными системами поддерживаются более длинные имена файлов переменной длины. Как это может быть реализовано?

Простейший метод состоит в установке ограничения на длину имени файла, обычно 255 символов, и использовании одной из схем, показанных на рис. 6.13. Такой способ прост, но он расходует много места в каталоге, так как длинные имена обычно бывают далеко не у всех файлов. Следовательно, для более эффективного использования дискового пространства желательно использовать другую структуру.

Один из альтернативных подходов состоит в отказе от предположения о том, что все записи в каталоге должны иметь один и тот же размер. При таком подходе каждая запись в каталоге начинается с порции фиксированного размера, обычно начинающейся с длины записи, за которой следуют данные в фиксированном формате — идентификатор владельца, дата создания, информация о защите и прочие атрибуты. Следом за заголовком фиксированной длины идет часть записи переменной длины, содержащая имя файла (рис. 6.14, *а*). На рисунке показаны три описателя файла, *project-budget*, *personnel* и *foo*. Имя каждого файла завершается специальным символом (обычно 0), обозначенным на рисунке перечеркнутыми квадратиками. Чтобы каждая запись в каталоге могла начинаться с границы слова, имя каждого файла дополняется до целого числа слов байтами, показанными на рисунке затененными прямоугольниками.

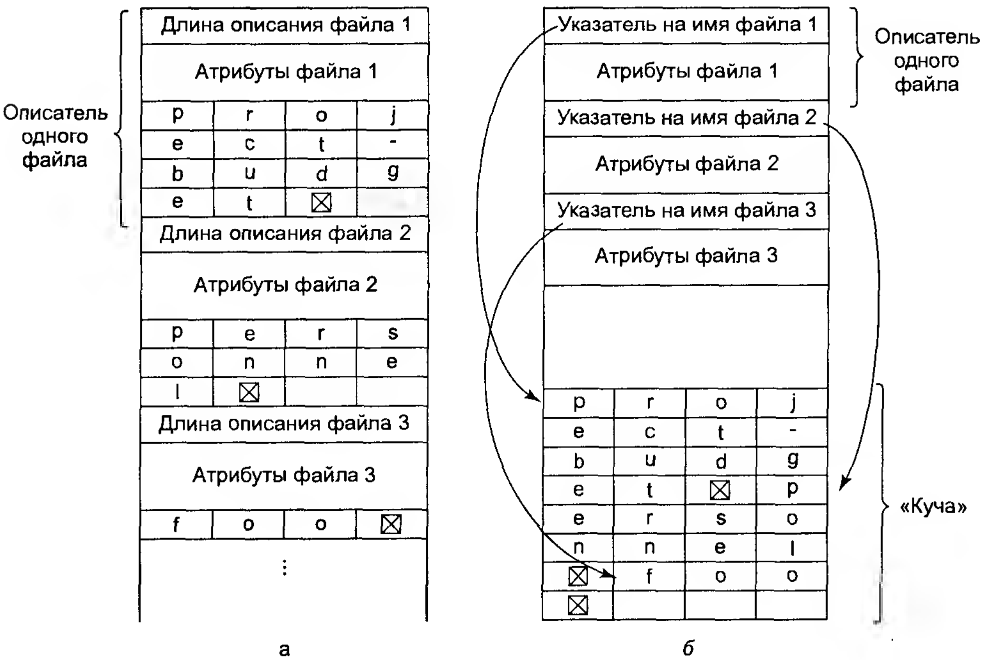


Рис. 6.14. Два варианта реализации длинных имен: прямо в записи каталога (*а*); в «куче» (*б*)

Недостаток этого метода состоит в том, что при удалении файла в каталоге остается промежуток переменной длины, в который описатель следующего файла может не поместиться. Эта проблема аналогична проблеме хранения на диске непрерывных файлов, хотя уплотнить каталог значительно легче, чем весь диск. Другая проблема связана с тем, что каталоговые записи переменной длины могут

занимать сразу две страницы памяти. При чтении такой каталоговой записи может возникнуть прерывание из-за отсутствия в оперативной памяти следующей страницы.

Другой метод реализации длинных имен файлов заключается в том, чтобы сделать все записи каталога фиксированной (равной) длины и хранить в них только указатели на имена, а сами имена хранить отдельно в «куче», в конце каталога, как показано на рис. 6.14, *б*. Преимущество этого метода состоит в том, что при удалении файла освободившееся место в каталоге точно подойдет для нового описателя файла. Тем не менее «кучу» придется все так же «разгребать», и при чтении длинного имени из нее также может возникнуть прерывание из-за отсутствия страницы памяти. Однако имена файлов уже не должны начинаться с границы слов, поэтому символы-заполнители не потребуются.

Во всех рассмотренных нами пока схемах при поиске файла каталоги просматриваются линейно сверху вниз. Для очень больших каталогов, содержащих много тысяч файлов, такой поиск может занять довольно много времени. Один из способов ускорить поиск файла состоит в использовании хэш-таблицы в каждом каталоге. Пусть размер такой таблицы будет равен n. При добавлении в каталог нового файла его имя должно хэшироваться в число от 0 до $n-1$. В качестве хэш-функции может использоваться, например, взятие остатка от деления имени файла на n. В качестве альтернативы можно делить не само имя, а сумму слов, его образующих. Возможны и другие варианты.

В любом случае исследуется элемент таблицы, соответствующий полученному хэш-коду. Если элемент не используется, туда помещается указатель на описатель файла. (Описатели файлов размещаются вслед за хэш-таблицей.) Если же элемент таблицы уже занят, то создается связный список, объединяющий все описатели файлов с одинаковым хэш-кодом.

Поиск файла осуществляется аналогично. Имя файла хэшируется. По хэш-коду определяется элемент таблицы. Затем проверяются все описатели файла из связного списка и сравниваются с искомым именем файла обычным способом. Если имени файла в связном списке нет, это означает, что файла нет в каталоге.

Преимуществом использования хэш-таблицы является ускоренный в несколько раз поиск файла. Недостаток этого метода состоит в более сложном администрировании каталога. Применять этот метод стоит только в тех системах, в которых ожидается, что каталоги будут содержать сотни и тысячи файлов.

Принципиально отличный способ ускорения процесса поиска файлов в больших каталогах заключается в кэшировании результатов поиска. Прежде чем начать поиск файла, проверяется, нет ли его имени в кэше. Если файловая система недавно уже искала этот файл, его имя окажется в кэше и повторная операция поиска будет выполнена очень быстро. Конечно, кэширование поможет только в том случае, если файловая система много раз обращается к небольшому количеству файлов.

Совместно используемые файлы

Когда несколько пользователей одновременно работают над одним проектом, им часто бывает нужно совместно использовать одни и те же файлы. Поэтому часто оказывается удобным, чтобы совместно используемый файл одновременно при-

сутствовал в различных каталогах, принадлежащих различным пользователям. На рис. 6.15 снова показана файловая система с рис. 6.6, только теперь один из файлов пользователя *C* присутствует также в каталоге пользователя *B*. Соединение между каталогом пользователя *B* и совместно используемым файлом называется **связью**. Сама файловая система теперь представляет собой **ориентированный ациклический граф** (DAG, Directed Acyclic Graph), а не дерево.

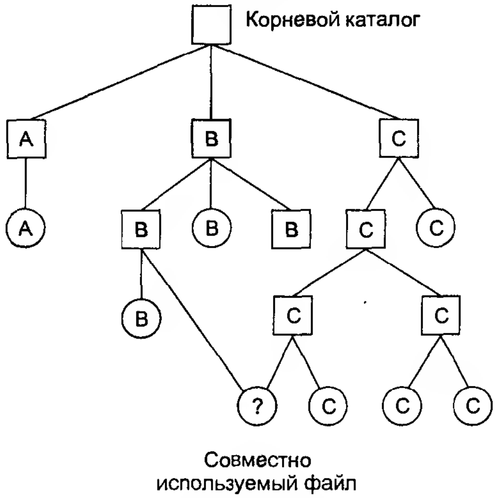


Рис. 6.15. Файловая система, содержащая совместно используемый файл

Совместное использование файлов удобно, но в то же время создает некоторые новые проблемы. Если дисковые адреса содержатся в самих каталоговых записях, тогда при добавлении новых данных к совместно используемому файлу новые блоки будут числиться только в каталоге того пользователя, который производил эти изменения с файлом. Другим пользователям эти изменения будут не видны.

Эта проблема имеет два решения. Во-первых, информация о блоках диска, занимаемых файлом, может содержаться не в каталоге, а в специальной структуре данных, связанной с файлом. В этом случае записи в каталогах будут просто указывать на эту структуру данных. Этот метод используется в системе UNIX (структура данных называется там i-узлом).

Второе решение состоит в том, что когда пользователь *B* устанавливает связь с одним из файлов пользователя *C*, система создает в каталоге пользователя *B* новый файл типа LINK (связь). Новый файл содержит просто имя пути к файлу, с которым он связан. Когда пользователь *B* читает данные из связанного файла, операционная система видит, что обращение производится к файлу типа LINK, поэтому она открывает файл, с которым связан этот файл, и читает данные из него. Такой метод называется **символьным связыванием**.

У каждого из этих методов имеются недостатки. В первом методе, когда пользователь *B* устанавливает связь с совместно используемым файлом, в i-узле владельцем файла числится пользователь *C*. Создание связи не изменяет владельца файла (рис. 6.16), а только увеличивает счетчик i-узла, что позволяет системе отслеживать количество записей в каталогах, ссылающихся на этот файл.

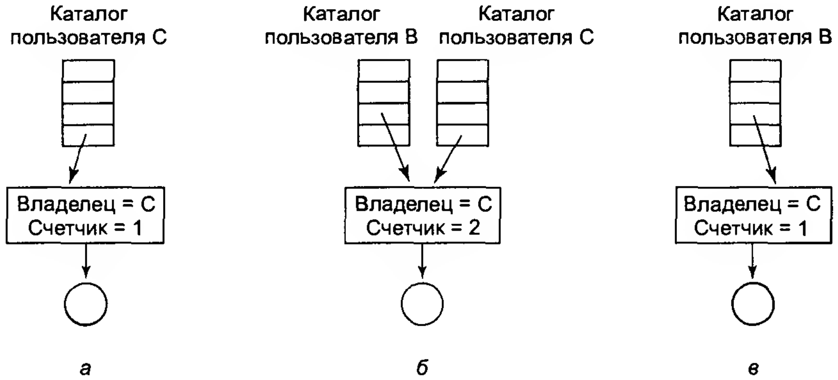


Рис. 6.16. Ситуация до связывания (а); после создания связи (*б*); владелец удалил свой файл (*в*)

Если впоследствии пользователь *С* попытается удалить этот файл, операционная система столкнется с проблемой. Если она удалит файл и очистит i-узел, у пользователя *В* окажется запись в каталоге, указывающая на неверный i-узел. Если этот i-узел впоследствии будет назначен другому файлу, связь в каталоге пользователя *В* будет ссылаться на неверный файл. По значению счетчика i-узла система сможет определить, что этот файл используется кем-то еще, но у нее нет способа найти все записи в каталогах, указывающие на этот файл, чтобы удалить их. Указатели на каталоги не могут храниться в i-узлах, так как число таких каталогов ничем не ограничивается.

Единственное, что может сделать операционная система — это удалить запись в каталоге пользователя *С*, но оставить на месте i-узел, уменьшив значение счетчика на 1, как показано на рис. 6.16, *в*. Таким образом, мы теперь получили ситуацию, в которой пользователь *В* является единственным пользователем, имеющим ссылку на файл, владельцем которого считается пользователь *С*. Если система ведет учет или выделяет ограниченные квоты на использование дискового пространства, пользователь *С* будет продолжать получать счета за этот файл до тех пор, пока пользователь *В* не решит, наконец, удалить его. В этом случае счетчик i-узла уменьшится до нуля и система удалит файл.

При символьном связывании такой проблемы не возникает, так как указатель на i-узел есть только у владельца файла. У остальных пользователей, установивших связь с этим файлом, есть только пути к файлу, а не указатели на i-узел. Когда *владелец* удаляет файл, файл уничтожается. Последующие попытки использовать этот файл при помощи символьной связи не будут иметь успеха, так как система не сможет найти файл. Удаление символьной связи никоим образом не повлияет на файл.

Недостатком символьной связи является необходимость накладных расходов. Чтобы получить доступ к i-узлу, следует сначала прочитать файл, содержащий путь. Затем следует пройти по этому пути, открывая каталог за каталогом, пока, наконец, не будет получен нужный i-узел. Все эти действия могут потребовать существенного количества обращений к диску. Кроме того, для каждой символьной связи требуется дополнительный i-узел, а также дополнительный блок диска

для хранения пути к файлу, хотя, если имя пути короткое, то в качестве оптимизации система может хранить его в самом i-узле. Преимущество символьных связей состоит в том, что они могут использоваться для ссылок на файлы, расположенные на удаленных компьютерах в любой точке земного шара. Для этого, кроме обычного пути файла, нужно всего лишь указать сетевой адрес машины, на которой файл располагается.

Использование связей, как символьных, так и других, создает еще одну проблему. При использовании связей у одного файла оказывается несколько путей. Программы, сканирующие каталоги со всеми подкаталогами, могут наткнуться на один и тот же файл несколько раз. Если такая программа, например, записывает все файлы на магнитофонную ленту, она запишет на ленту несколько копий одного и того же файла. Более того, если запись с этой ленты будет потом прочитана на другом компьютере, этот файл окажется скопированным на диск несколько раз, если только считывающая ленту программа не догадается, что это один и тот же файл.

Организация дискового пространства

Обычно файлы хранятся на диске, поэтому организация дискового пространства является основной заботой разработчиков файловой системы. Для хранения файла из *n* байтов возможно использование двух стратегий: выделение на диске *n* последовательных байтов или разбиение файла на несколько непрерывных блоков. Та же дилемма присутствует в системах управления памяти, где имеется выбор между чистой сегментацией и страничной организацией памяти.

Как уже было показано, при хранении файла в виде непрерывной последовательности байтов возникает проблема, связанная с увеличением его размеров. Единственный способ увеличить непрерывный файл состоит в перемещении его на новое место на диске. Проблема существенна и для управления сегментами памяти, с той разницей, что перемещение сегмента в памяти является более быстрой операцией по сравнению с перемещением файла на диске. По этой причине почти все файловые системы хранят файлы в виде блоков фиксированного размера, расположенных в различных частях диска.

Размер блока

Как только принято решение хранить файлы блоками фиксированного размера, возникает вопрос о размере этих блоков. Учитывая организацию дисков, очевидными кандидатами на роль сегментов файлов являются сектор, дорожка и цилиндр диска (минусом такого выбора является зависимость этих параметров от устройств). В системе управления страницами памяти размер страницы также входит в число основных претендентов.

Если выбрать большую единицу хранения, такую как цилиндр, это будет означать, что любой файл, даже состоящий из одного байта, займет как минимум целый цилиндр. Изучения показали, что средний размер файла в системе UNIX около 1 Кбайт, так что при выделении каждому файлу 32-килобайтного блока будет расходоваться понапрасну 31/32 или 97 % общего дискового пространства [241].

С другой стороны, при использовании маленьких единиц хранения каждый файл будет состоять из большого числа блоков. Для чтения каждого блока файла

обычно требуется операция поиска нужного цилиндра и задержка вращения диска, поэтому чтение файла, состоящего из большого числа блоков, будет медленным.

Например, представьте себе диск с 131 072 байтами (128 Кбайт) на дорожку, периодом вращения 8,33 мс и средним временем поиска 10 мс. При этом время, требующееся для чтения блока из *k* байтов, будет равно сумме времени поиска, задержки вращения и времени переноса данных:

```
10 + 4.165 + (k/131072) × 8.33
```

Жирная ломаная линия на рис. 6.17 показывает зависимость скорости передачи данных от размера блока. Чтобы вычислить эффективность использования дискового пространства, нам нужно сделать предположение о среднем размере файла. Недавние измерения, проведенные автором на факультете, где около 1000 пользователей компьютеров и более миллиона дисковых файлов системы UNIX, показали, что медиановый размер файла равен 1680 байт. Это означает, что половина файлов меньше 1680 байт, а другая половина больше этого размера. Следует заметить, что медиановый размер представляет собой лучшую единицу измерения, чем средний, так как небольшое количество файлов может очень сильно повлиять на среднее значение, но не на медиану. (Среднее значение оказалось равным 10 845 байт благодаря нескольким 100-мегабайтным руководствам по аппаратуре, которые оказались в момент измерений на дисках). Для простоты предположим, что все файлы имеют размер, равный 2 Кбайт, в результате чего мы получим штриховую линию, изображающую на рис. 6.17 эффективность использования дискового пространства.

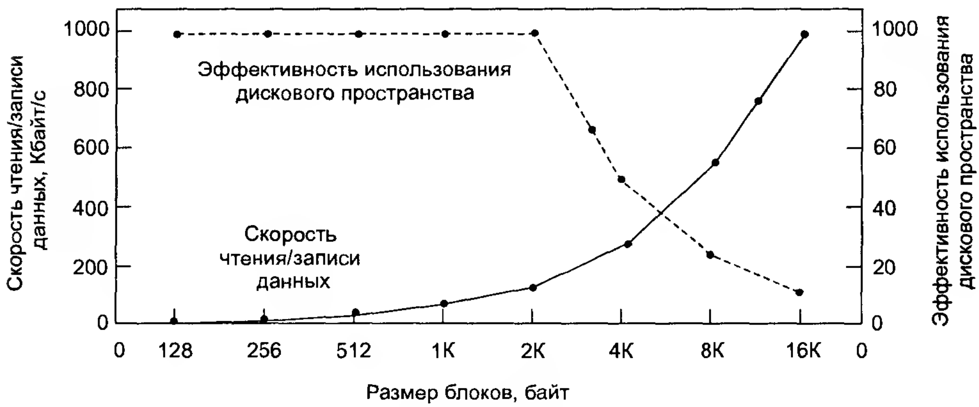


Рис. 6.17. Зависимость скорости чтения/записи данных диска (жирная линия, левая шкала) и эффективности использования дискового пространства (штриховая линия, правая шкала) от размера блоков. Все файлы по 2 Кбайт

Эти две кривые можно понимать следующим образом. Время доступа к блоку практически полностью определяется временем поиска и задержкой вращения, поэтому, если считать, что для доступа к блоку требуется 14 мс, чем больше данных удастся считать за одну такую операцию, тем лучше. Таким образом, скорость чтения/записи данных растет пропорционально размеру блока до тех пор, пока блок не вырастет настолько, что важнее окажется время передачи блока. При

использовании небольших блоков, размер которых составляет степень числа два, и 2-килобайтных файлов потерь дискового пространства нет. Однако при хранении 2-килобайтных файлов в 4-килобайтных блоках теряется уже 50 % дискового пространства. В действительности мало файлов имеют длину, кратную размеру блока, поэтому какая-то часть дискового пространства всегда теряется в последнем блоке файла.

Что показывает данный график? То, что производительность и использование дискового пространства являются взаимоисключающими целями. При небольших блоках низкая производительность, зато хорошее использование дискового пространства, и наоборот. Требуется найти некий компромиссный размер. Для подобных данных 4 Кбайт может оказаться оптимальным выбором, однако некоторые операционные системы сделали свой выбор много лет назад, когда и параметры дисков, и размеры файлов были другими. В системе UNIX обычно используется размер блока 1 Кбайт. В системе MS-DOS размер блока может быть любой степенью двух от 512 байт до 32 Кбайт и определяется размером диска (так как максимальное количество блоков в разделе диска равно 2^{16} и размер блока оказывается пропорциональным размеру диска).

Пытаясь определить, является ли использование файлов в Windows NT существенно отличным от их употребления в UNIX, Фогельс в университете Корнелла произвел соответствующие измерения [346]. Он пришел к выводу, что использование файлов в NT более сложное, чем в UNIX. Он писал:

> *Когда мы вводим несколько символов в текстовом редакторе* notepad, *сохраняя их в файле, мы запускаем 26 системных вызовов, включая 3 неудачные попытки открытия файлов, перезапись одного файла и 4 дополнительных последовательности операций открытия и закрытия файлов.*

Тем не менее его измерения показали медиановый размер (взвешенный по использованию) только читаемых файлов, равный 1 Кбайт, только записываемых файлов — 2,3 Кбайт, и файлов, читаемых и записываемых, равный 4,2 Кбайт. Учитывая, что университет Корнелла производил значительно больше масштабных научных вычислений, чем учреждение автора, а также различие в технике измерений (статической и динамической), полученные результаты в достаточной мере согласуются с медиановым размером файлов около 2 Кбайт.

Учет свободных блоков

После того как мы выбрали размер блоков, следует определить, как учитывать свободные и занятые блоки. Широкое применение получили два метода, показанные на рис. 6.18. Первый метод представляет собой использование связного списка блоков диска. При этом в каждом блоке списка содержится столько номеров свободных блоков, сколько может поместиться в один блок. При размере блока, равном 1 Кбайт, и 32-разрядных номерах блоков каждый блок списка свободных блоков может содержать номера 255 свободных блоков. (Одно 32-разрядное слово нужно для указателя на следующий блок списка). Для 16-гигабайтного диска потребуется список свободных блоков, состоящий из 16 794 блоков, чтобы содержать все 2^{24} номера дисковых блоков. Часто для списка резервируется нужное число блоков в начале диска.

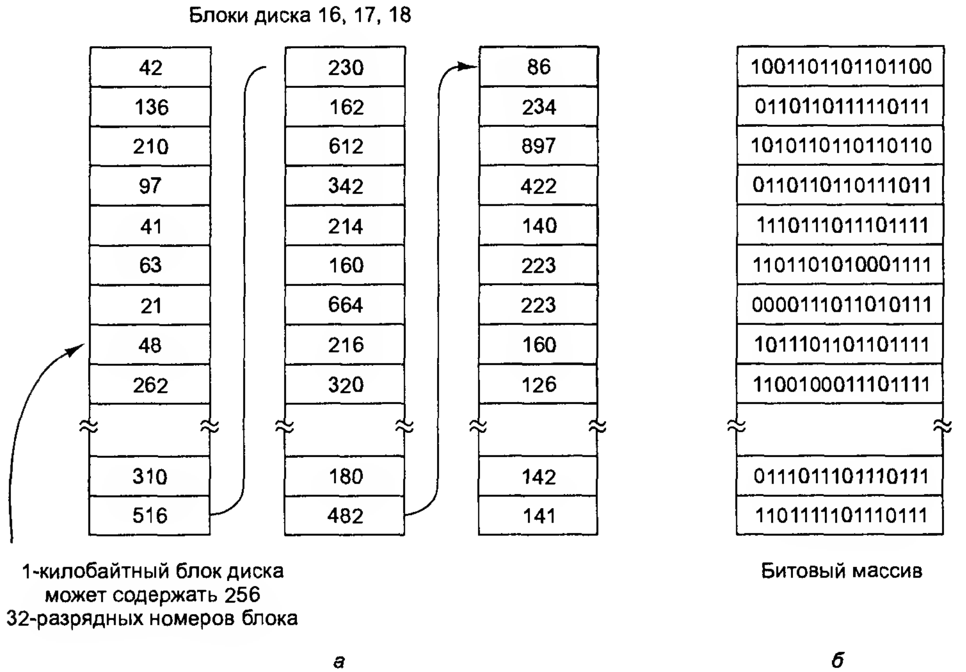
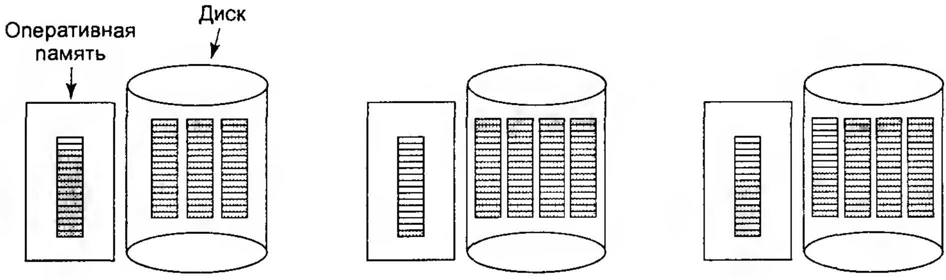


Рис. 6.18. Хранение информации о свободных блоках в виде списка (*а*); битовый массив (*б*)

Другой метод учета свободного дискового пространства состоит в хранении этой информации в виде битового массива (иногда называемого бит-картой). При этом на каждый блок приходится всего по одному биту вместо 32. Свободные блоки обозначаются в массиве единицами, а занятые — нулями (или наоборот). 16-гигабайтный диск состоит из 2^{24} килобайтных блоков, таким образом, для него требуется массив размером 2^{24} бит, то есть 2048 блоков.

При использовании метода учета свободных блоков при помощи списка в оперативной памяти требуется хранить только один блок указателей. Когда создается файл, нужные блоки берутся из блока указателей. Когда указатели в этом блоке заканчиваются, читается новый блок с диска. Соответственно, при удалении файла его блоки освобождаются, а их номера добавляются в список, хранящийся в оперативной памяти. Когда блок указателей наполняется, он записывается на диск.

При определенных обстоятельствах такой метод приводит к излишним операциям ввода-вывода. Рассмотрим ситуацию на рис. 6.19, *а*, в которой в блоке указателей есть место только для еще двух записей (затененные прямоугольники обозначают указатели в блоке). При удалении трехблочного файла блок указателей переполняется и его нужно записать на диск, что приводит к ситуации, показанной на рис. 6.19, *б*. Если теперь снова создается трехблочный файл, полный блок указателей должен быть снова прочитан с диска, что нас возвращает к ситуации на рис. 6.19, *а*. Если этот трехблочный файл был временным файлом, то он вскоре опять удаляется, при этом опять блок указателей пишется на диск. Таким образом, когда указатели в блоке оказываются на границе блока, возникает необходимость в дополнительных операциях ввода-вывода.



а б в

Рис. 6.19. Почти полный блок указателей свободных блоков в памяти и три блока указателей на диске (*а*); результат удаления трехблочного файла (*б*); альтернативная стратегия (*в*)

Существует метод, позволяющий избежать лишних операций ввода-вывода. Он состоит в том, что полный блок указателей расщепляется на два. При этом из ситуации на рис. 6.19, *а* вместо ситуации на рис. 6.19, *б* мы переходим к ситуации на рис. 6.19, *в*. При заполнении блока указателей в памяти этот блок записывается на диск не в виде заполненного до конца блока, а как блок, заполненный наполовину, то есть записывается, по сути, половина указателей блока. В памяти остается блок с остальными указателями, то есть также заполненный наполовину блок. Таким образом, хранящийся в оперативной памяти блок указателей максимально готов как к добавлению в него новых указателей, так и к освобождению хранящихся в нем.

При использовании битового массива также возможно хранение в памяти всего одного блока с обращением к диску за другим блоком, когда текущий блок переполняется или, наоборот, становится пустым. Дополнительное преимущество такого подхода состоит в том, что выделяемые файлу блоки будут располагаться близко друг к другу, в результате чего для доступа к файлу будет затрачено меньше времени на перемещение блока головок. Поскольку битовый массив представляет собой структуру данных фиксированного размера, то при (частичной) постраничной организации ядра битовый массив может быть помещен в виртуальную память, откуда можно получать страницы по мере надобности.

Дисковые квоты

Чтобы не допустить захвата пользователями слишком больших участков дискового пространства, многопользовательские операционные системы часто содержат механизм предоставления дисковых квот. Суть в том, что администратор назначает каждому пользователю максимальную долю файлов и блоков, а операционная система гарантирует, что пользователи не превышают выделенных им квот. Типичный механизм реализации этой идеи описан ниже.

Когда пользователь открывает файл, операционная система находит его атрибуты и дисковые адреса и помещает их в таблицу в оперативной памяти. Среди атрибутов находится элемент, сообщающий, кто является владельцем данного файла. Любые увеличения размеров файла будут учитываться в записи квоты для этого пользователя.

Вторая таблица содержит запись квоты для каждого пользователя, файл которого открыт в данный момент, даже если этот файл был открыт кем-либо другим.

Эта таблица показана на рис. 6.20. Она извлекается из файла квот, хранящегося на диске. Когда все файлы закрыты, запись записывается обратно в файл.

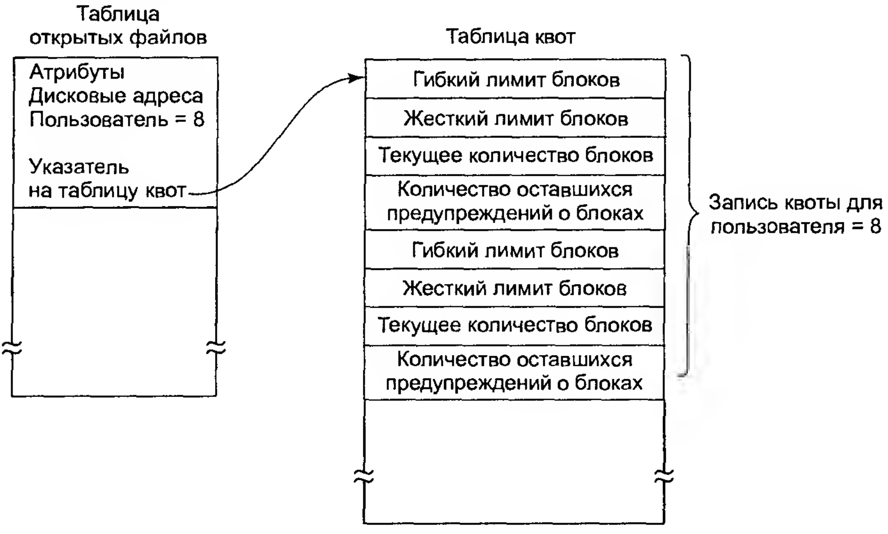


Рис. 6.20. Квоты учитываются для каждого пользователя в таблице квот

Когда в таблице открытых файлов создается новый элемент, в него помещается указатель на запись квоты владельца файла. При каждом добавлении нового блока к файлу общее количество блоков, числящееся за пользователем, увеличивается и сравнивается с гибким и жестким лимитом. Гибкий лимит может быть превышен, но жесткий нет. При достижении жесткого лимита любая попытка добавить блок к файлу будет завершаться ошибкой. Аналогично проверяется количество файлов пользователя.

Когда пользователь пытается зарегистрироваться в системе, операционная система считывает его файл квот, проверяя, не превысил ли пользователь гибкие пределы по числу блоков или числу файлов. Если один из пределов превышен, операционная система выдает предупреждение, а счетчик оставшихся предупреждений уменьшается на единицу. Когда счетчик предупреждений достигает нуля, операционная система отказывает пользователю в регистрации. Чтобы снова получить возможность входить в систему, пользователю необходимо обратиться к системному администратору.

Таким образом, пользователь может превысить свои гибкие лимиты, при условии, что перед выходом из системы он удалит лишние файлы, превышающие эти пределы. Жесткие лимиты превышены быть не могут.

Надежность файловой системы

Разрушение файловой системы часто оказывается большим бедствием, чем поломка компьютера. Если компьютер разрушается вследствие пожара, удара молнии или пользователь прольет чашку кофе на клавиатуру, это неприятно, ремонт потребует денег, но обычно не причинит много хлопот. Недорогие персональные

компьютеры можно заменить в течение часа, обратившись к соответствующему коммерсанту. (Исключение составляют университеты, в которых для приобретения персональных компьютеров требуется согласование этого вопроса в трех комитетах, получение пяти подписей и 90 дней ожидания.)

В случае же потери файловой системы по вине аппаратуры, или программного обеспечения, или крыс, прогрызших дополнительные отверстия в гибких дисках, восстановление всей информации будет трудным, требующим много времени, а часто и невозможным делом. Для пользователей, чьи программы, документы, файлы клиентов, счета, базы данных, маркетинговые планы или другие данные утеряны навсегда, последствия могут оказаться катастрофическими. Хотя операционная система не может защитить от физического уничтожения оборудования или носителя, она может помочь сберечь информацию. В данном разделе мы рассмотрим некоторые вопросы, касающиеся защиты файловой системы от уничтожения.

Когда гибкие диски покидают фабрику, их качество, как правило, превосходно, но со временем на них могут появиться дефектные блоки. У жестких дисков дефектные блоки часто бывают с самого начала, так как производство жестких дисков абсолютно без дефектных блоков слишком дорого. Как уже рассказывалось в главе 5, дефектные блоки обычно контроллер заменяет специальными запасными блоками. Тем не менее эта техника не способна защитить абсолютно ото всех возможных вариантов потери информации.

Резервные копии

Большинство пользователей считает создание резервных копий файлов просто потерей времени. Однако когда в один прекрасный день их диск внезапно приказывает долго жить, они диаметрально меняют свои привычки. Компании, напротив, обычно хорошо осознают ценность своей информации и создают резервные копии файлов один раз в сутки, чаще всего на магнитной ленте. Современные магнитные ленты вмещают десятки и иногда даже сотни гигабайт при цене в несколько центов за гигабайт. Тем не менее создание резервных копий является далеко не столь тривиальным делом, как это может показаться, поэтому мы ниже рассмотрим некоторые аспекты данной темы.

Резервные копии на магнитной ленте обычно создаются для возможного восстановления информации при потенциальном ее уничтожении вследствие аварии или ошибки пользователя. К авариям можно отнести такие события, как выход из строя жесткого диска, пожары, потопы и другие природные катаклизмы. На практике такое случается нечасто, поэтому пользователи редко беспокоятся о создании резервных копий. Как правило, эти пользователи по той же причине не страхуют свое имущество от пожаров и прочих стихийных бедствий.

Вторая причина состоит в том, что пользователи часто случайно удаляют файлы, которые позднее оказываются нужными. Эта проблема возникает столь часто, что в системе Windows при обычном удалении файла он на самом деле не удаляется, а помещается в специальный каталог, называемый **мусорной корзиной**, откуда его при необходимости несложно выудить обратно. Резервные копии продолжают этот принцип дальше, позволяя восстанавливать с архивных лент файлы, удаленные много дней и даже недель назад.

Создание резервных копий занимает много времени и требует много места, поэтому особенное значение при этом получают эффективность и удобство. Во-первых, следует ли архивировать всю файловую систему или только ее часть? Многие операционные системы хранят двоичные файлы в отдельных каталогах. Эти файлы не обязательно архивировать, так как они могут быть переустановлены с CD-ROM производителя. Кроме того, большинство систем имеет каталог для временных файлов. В их архивировании также нет необходимости. В системе UNIX все специальные файлы (устройств ввода-вывода) хранятся в каталоге */dev*. Создавать резервную копию этого каталога не только ненужно, но и опасно, так как программа архивации может навсегда повиснуть, если попытается читать эти файлы. Поэтому обычно создаются резервные копии отдельных каталогов, а не всей файловой системы.

Во-вторых, архивировать не изменившиеся с момента последней архивации файлы неэффективно, поэтому на практике применяется идея **инкрементных резервных копий**, или, как их еще называют, инкрементных дампов. Простейшая форма инкрементного архивирования состоит в том, что полная резервная копия создается, скажем, раз в неделю или раз в месяц, а ежедневно сохраняются только те файлы, которые изменились с момента последней полной архивации. Еще лучше архивировать только те файлы, которые изменились с момента последней архивации. Такая схема сокращает время создания резервных копий, но усложняет процесс восстановления, так как для этого нужно сначала восстановить файлы последней полной архивации, а затем проделать ту же процедуру со всеми инкрементными резервными копиями. Чтобы упростить восстановление, часто применяются более сложные схемы инкрементной архивации.

В-третьих, поскольку объем архивируемых данных обычно очень велик, желательно сжимать эти данные до записи на магнитную ленту. Однако многие алгоритмы сжатия данных устроены так, что малейший дефект ленты может привести к нечитаемости всего файла или даже всей ленты. Поэтому следует хорошенько подумать, прежде чем принимать решение о сжатии архивируемых данных.

В-четвертых, создание резервной копии сложно выполнять в активной файловой системе. Если во время архивации создаются, удаляются и изменяются файлы и каталоги, то информация в создаваемом архиве может оказаться противоречивой. В то же время, поскольку создание резервной копии может занять несколько часов, для этого может понадобиться отключение системы на большую часть ночи, что не всегда приемлемо. По этой причине были разработаны алгоритмы, способные быстро фиксировать состояние файловой системы для ее архивации, копируя критические структуры данных. Последующие изменения файлов и каталогов требовали вместо их замены дополнительного копирования отдельных блоков [160]. Таким образом, файловая система как бы замораживается в момент фиксации и может архивироваться позднее.

Наконец, в-пятых, создание резервных копий создает множество нетехнических проблем. Лучшая система безопасности, охраняющая информацию, может оказаться бесполезной, если системный администратор хранит все свои магнитные ленты с резервными копиями в неохраняемом помещении, которое еще и оставляет открытым. Все, что нужно сделать шпиону, это заскочить на секунду в комнату,

положить кассету в карман и не спеша покинуть здание. Прощай, безопасность. Кроме того, ежедневная архивация принесет мало пользы, если при пожаре погибнут не только компьютеры, но и ленты с резервными копиями. По этой причине резервные копии следует хранить в удаленном месте, в результате чего поддерживать безопасность на достаточно высоком уровне становится еще сложнее. Детальное обсуждение этого и других административных вопросов приводится в [244]. Ниже мы обсудим только технические аспекты создания резервных копий файловой системы.

Для создания резервной копии диска на ленте существует две стратегии: физическая архивация и логическая архивация. **Физическая архивация** состоит в поблочном копировании на магнитную ленту всего диска с блока 0 по последний блок. Эта программа настолько проста, что, возможно, она даже может быть полностью отлажена, чего нельзя сказать об остальных полезных программах.

И все же о физической архивации следует сказать несколько слов. Во-первых, в копировании неиспользуемых блоков диска мало пользы. Если программа архивации сможет получить доступ к структуре данных, хранящей информацию о свободных и занятых блоках диска, она может избежать копирования неиспользуемых блоков. Однако тогда придется перед каждым блоком записывать его номер.

Вторая проблема возникает при столкновении программы архивации с дефектными блоками диска. Если все дефектные блоки контроллер автоматически заменяет запасными блоками, как было описано в разделе «Обработка ошибок» главы 5, тогда физическая архивация работает прекрасно. Однако если дефектные блоки видны операционной системе, которая учитывает их в одном или нескольких специальных файлах или битовых массивах, то абсолютно необходимо, чтобы программа архивации могла получить доступ к этой информации во избежание ошибок чтения диска.

Главное преимущество физической архивации состоит в ее простоте и высокой скорости (обычно она может работать со скоростью диска). Основными ее недостатками являются неспособность пропускать определенные каталоги, производить инкрементную архивацию и восстанавливать отдельные файлы. По этим причинам в большинстве систем применяется логическая архивация.

Логическая архивация сканирует один или несколько указанных каталогов со всеми их подкаталогами и копирует на ленту все содержащиеся в них файлы и каталоги, изменившиеся с указанной даты (например, с момента последней архивации). Таким образом, при логической архивации на магнитную ленту записываются последовательности детально идентифицированных каталогов и файлов, что позволяет восстановить отдельный файл или каталог.

Поскольку логическая архивация применяется на практике чаще физической, познакомимся более детально с алгоритмом архивации на примере, показанном на рис. 6.21. Этот алгоритм используется в большинстве систем UNIX. На рисунке показано дерево файлов с каталогами (квадраты) и файлами (кружки). Затененные элементы были изменены с момента последней архивации и поэтому следует создать их резервную копию. Светлые элементы не нуждаются в архивации. Каждый каталог и файл помечены номером своего i-узла.

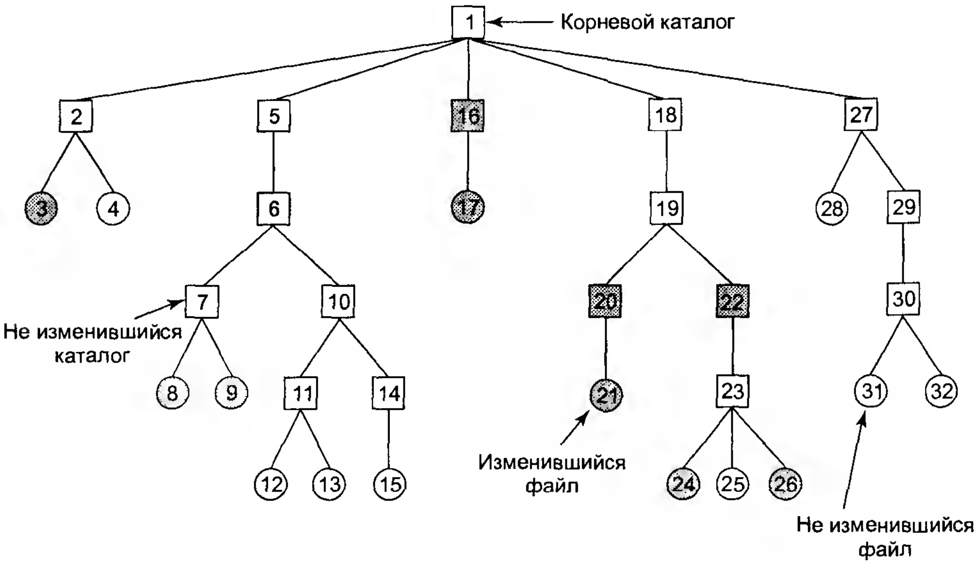


Рис. 6.21. Архивируемая файловая система

Этот алгоритм также создает резервные копии всех каталогов (даже не модифицированных), расположенных на пути к модифицированному файлу или каталогу. Делается это по двум причинам. Во-первых, чтобы все сохраненные файлы и каталоги можно было восстановить на другом компьютере. Благодаря этому программа архивации может использоваться для переноса всей файловой системы с одного компьютера на другой.

Во-вторых, сохранение резервных копий всех каталогов, расположенных на пути к модифицированному файлу, позволяет восстановить один отдельный файл (например, удаленный по ошибке). Представьте, что полная архивация системы произведена в воскресенье, а в понедельник выполнена инкрементная архивация. Во вторник удаляется каталог */usr/jhs/proj/nr3* со всеми содержащимися в нем каталогами и файлами. В среду утром пользователь хочет восстановить файл */usr/jhs/proj/nr3/plans/summary*. Однако нельзя просто восстановить файл *summary*, так как каталоги */usr/jhs/proj/nr3/plans* и */usr/jhs/proj/nr3* удалены и файл *summary* некуда поместить. Сначала нужно восстановить каталоги *nr3* и *plans*. Чтобы операционная система смогла корректно установить такие параметры этих каталогов, как идентификатор владельца, режимы доступа, время создания, время изменения и т. д., эти каталоги должны присутствовать на архивной ленте, несмотря на то, что сами каталоги не модифицировались с момента последней архивации.

Алгоритм архивации создает битовый массив, индексированный по номеру i-узла, в котором на каждый i-узел отводится несколько битов. Работа алгоритма протекает в четыре этапа. Первый этап начинается в начальном каталоге (например, в корневом), в котором исследуются все элементы. Для каждого модифицированного файла его i-узел помечается в битовом массиве. Каждый каталог также помечается (независимо от того, был он изменен или нет), после чего он открывается и рекурсивно исследуется все его содержимое.

К концу фазы 1 все модифицированные файлы и все каталоги помечаются в битовом массиве, как показано (затенением) на рис. 6.22, *а*. На втором этапе алгоритм снова рекурсивно прошагивает весь каталог, снимая отметку со всех каталогов, в которых нет модифицированных файлов или каталогов. В результате этих действий битовый массив принимает вид, показанный на рис. 6.22, *б*. Обратите внимание, что каталоги 10, 11, 14, 27, 29 и 30 теперь уже не помечены, так как в них и в их подкаталогах не содержится ничего модифицированного. Для этих каталогов не будет создаваться резервная копия. Напротив, каталоги 5 и 6 будут заархивированы, несмотря на то, что сами они не были модифицированы. Однако они потребуются, чтобы сегодняшние изменения можно было восстановить на новой машине. Для большей эффективности фазы 1 и 2 можно объединить, выполнив их за один проход.

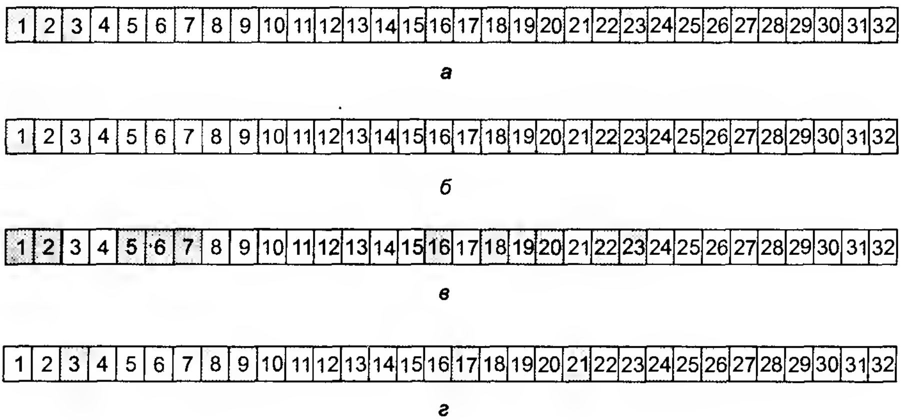


Рис. 6.22. Битовые массивы, используемые алгоритмом логической архивации

К этому моменту алгоритму известно, какие каталоги и файлы должны архивироваться (помеченные на рис. 6.22, *б*). Фаза 3 состоит из сканирования i-узлов по порядку номеров и создания резервных копий всех каталогов, помеченных для архивации. Они помечены на рис. 6.22, *в*. Перед каждым каталогом записываются его атрибуты (владелец, времена и т. д.), необходимые для восстановления. Наконец, на четвертом этапе файлы, помеченные на рис. 6.22, *г*, также архивируются со своими атрибутами, записываемыми перед ними. На этом архивация завершается.

Восстановление файловой системы происходит достаточно просто. Сначала на диске создается пустая файловая система. Затем восстанавливаются данные последней полной архивации. Сначала восстанавливаются каталоги, создавая скелет файловой системы, после чего восстанавливаются все файлы. Затем весь процесс повторяется со всеми инкрементными архивациями по очереди.

Хотя алгоритм логической архивации несложен, в этом деле имеется несколько хитростей. Во-первых, поскольку список свободных блоков не является файлом, он не архивируется, следовательно, его приходится восстанавливать с нуля, после того как были восстановлены все архивы[1]. Это всегда является выполнимой

[1] Точнее, никаких специальных действий для коррекции таблицы свободных блоков производить не нужно. Восстанавливаемые каталоги и файлы просто создаются заново (тем более что блоки, на которых они располагались ранее, уже могут оказаться заняты), после чего нужно лишь скорректировать их атрибуты. — *Примеч. перев.*

задачей, так как множество свободных блоков представляет собой всего лишь совокупность всех блоков, содержащихся во всех файлах.

Второй проблемой являются связи файлов. Если у файла имеются связи в двух или более каталогах, важно, чтобы этот файл был восстановлен только один раз, и чтобы во всех каталогах, в которых были ссылки на этот файл, появились именно ссылки.

Еще одна проблема состоит в том, что в системе UNIX файлы могут содержать дыры. Считается допустимым открыть файл, записать в него несколько байтов, затем переместить указатель в файле на много блоков в глубь файла, после чего записать еще несколько байтов. Блоки посередине не являются частью файла и не должны архивироваться и восстанавливаться. Файлы, содержащие образы памяти, часто содержат большие пустые пространства между сегментом данных и стеком. Если их не обрабатывать должным образом, то каждый восстановленный файл с образом памяти заполнит эту область нулями и окажется того же размера, что и виртуальное адресное пространство (например, 2^{32} байт или, что еще хуже 2^{64} байт).

Наконец, специальные файлы, называемые каналами или трубами и т. п., никогда не должны архивироваться, независимо от того, в котором каталоге они могут оказаться (они не обязаны находиться в каталоге */dev*). Дополнительные сведения о файловой системе см. в [67, 372].

Непротиворечивость файловой системы

Еще одним аспектом, относящимся к проблеме надежности, является непротиворечивость файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии. Эта проблема становится особенно важной в случае, если одним из модифицированных и не сохраненных блоков оказывается блок i-узла, каталога или списка свободных блоков.

Для решения проблемы противоречивости файловой системы на большинстве компьютеров имеется специальная обслуживающая программа, проверяющая непротиворечивость файловой системы. Например, в системе UNIX такой программой является *fsck*, а в системе Windows это программа *scandisk*. Эта программа может быть запущена сразу после загрузки системы, особенно если до этого произошел сбой. Ниже будет описано, как работает утилита *fsck*. Утилита *scandisk* несколько отличается от *fsck*, поскольку работает в другой файловой системе, однако для нее также остается верным принцип использования избыточной информации для восстановления файловой системы. Все программы проверки файловой системы проверяют различные файловые системы (дисковые разделы) независимо друг от друга.

Существует два типа проверки непротиворечивости: блоков и файлов. При проверке непротиворечивости блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный на 0. Счетчики в первой таблице учитывают, сколько раз каждый блок присутствует в файле. Счетчики во второй таблице записывают, сколько раз каждый блок учитывается в списке свободных блоков (или в битовом массиве свободных блоков).

Затем программа считывает все i-узлы. Начиная с i-узла, можно построить список всех номеров блоков, используемых в соответствующем файле. При считывании каждого номера блока соответствующий ему счетчик увеличивается на единицу. Затем программа анализирует список или битовый массив свободных блоков, чтобы обнаружить все неиспользуемые блоки. Каждый раз, встречая номер блока в списке свободных блоков, программа увеличивает на единицу соответствующий счетчик во второй таблице.

Если файловая система непротиворечива, то каждый блок будет встречаться только один раз, либо в первой, либо во второй таблице, как показано на рис. 6.23, *а*. Однако в результате сбоя эти таблицы могут принять вид, показанный на рис. 6.23, *б*. В этом случае блок два отсутствует в каждой таблице. О таком блоке программа сообщит как о **недостающем блоке**. Хотя пропавшие блоки не причиняют вреда, они занимают место на диске, снижая его емкость. Решить проблему пропавших блоков очень просто: программа проверки файловой системы просто добавляет эти блоки к списку свободных блоков.

Номер блока

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Занятые блоки
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Свободные блоки

а

Номер блока

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Занятые блоки
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	Свободные блоки

б

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Занятые блоки
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Свободные блоки

в

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Занятые блоки
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Свободные блоки

г

Рис. 6.23. Состояния файловой системы: непротиворечивое (*а*); пропавший блок (*б*); дубликат блока в списке свободных блоков (в); дубликат блока данных (*г*)

Другая возможная ситуация показана на рис. 6.23, *в*. Здесь мы видим блок номер 4, дважды появляющийся в свободном списке. (Дубликаты свободных блоков возможны только в том случае, если файловая система действительно применяет списки свободных блоков; при использовании битового массива такая ситуация невозможна.) Решение этой проблемы также несложно: построить список свободных блоков заново.

Гораздо хуже ситуация, в которой один и тот же блок окажется сразу в двух файлах, как показано на рис. 6.23, *г* в случае с блоком 5. При удалении любого из этих файлов блок 5 будет помещен в список свободных блоков, что приведет к ситуации, в которой один и тот же блок одновременно является и свободным и занятым. Если удалить оба файла, этот блок будет помещен в список свободных блоков дважды.

В такой ситуации программа проверки файловой системы должна взять свободный блок, скопировать в него содержимое блока 5 и вставить эту копию в один из файлов. Таким образом, содержимое файлов останется неизменным (хотя почти наверняка один из файлов уже испорчен), но, по крайней мере, структура файловой системы после этой операции становится непротиворечивой. Программа также должна выдать сообщение об ошибке, чтобы пользователь мог изучить повреждение.

Помимо проверки правильности принадлежности блоков программа проверки файловой системы также проверяет каталоговую структуру. Для этого также используется таблица счетчиков, но уже не для блоков, а для файлов. Проверка начинается с корневого каталога с рекурсивным заходом в каждый каталог. Для каждого файла в каждом каталоге программа увеличивает на единицу счетчик использования файла. Благодаря жестким связям файл может присутствовать сразу в нескольких каталогах. Символьные связи не учитываются и не увеличивают счетчик.

Когда сканирование дерева каталогов завершено, программа получает список, индексированный по номерам i-узлов, сообщающий, в скольких каталогах присутствует каждый файл. Затем программа сравнивает полученные числа со счетчиками связей, хранящимися в самих i-узлах. Эти счетчики содержат 1 при создании файла и увеличиваются на единицу каждый раз, когда создается связь (жесткая) с данным файлом. В непротиворечивой файловой системе оба счетчика должны совпадать. Однако возможны два типа ошибок: значение счетчика связи в i-узле может оказать слишком велико или слишком мало.

Если счетчик связи больше, чем количество каталоговых записей, тогда даже при удалении всех файлов из каталогов счетчик все равно не уменьшится до нуля и i-узел не будет удален. Эта ошибка не серьезная, но она приводит к расходованию дискового пространства файлом, не находящимся ни в одном каталоге. Чтобы исправить ее, следует установить значение счетчика равным числу существующих каталоговых записей.

Вторая ошибка таит в себе катастрофические последствия. Если у файла есть две каталоговые записи, связанные с ним, но в i-узле утверждается, что описатель у файла только один, тогда при удалении описателя этого файла в любом каталоге

счетчик i-узла уменьшится до нуля. При этом файловая система освободит все блоки, занимаемые файлом, в том числе и блок, в котором помещается сам i-узел. Таким образом, в одном из каталогов сохранится описатель файла, указывающий на неиспользуемый i-узел, чьи блоки могут быть вскоре выделены другим файлам. Решение также заключается в присваивании значения счетчика i-узла фактическому числу описателей файла.

Часто эти две операции, проверки блоков и проверки каталогов, для увеличения эффективности объединяют в один проход. Возможно также проведение и других проверок. Например, формат каталогов должен соответствовать определенным требованиям, с i-узлами и ASCII-именами. Если i-узел оказывается больше числа i-узлов на диске, это означает, что каталог поврежден.

Более того, у каждого i-узла могут оказаться значения режима доступа, являющиеся допустимыми, но странными, как, например, 0007, совсем отказывающий в доступе владельцу и его группе, но зато разрешающий всем посторонним пользователям читать, писать и исполнять файл. Программа должна хотя бы сообщать обо всех файлах, предоставляющих посторонним пользователям больше прав, чем владельцам. Каталоги, содержащие, скажем, более 1000 описателей файлов, также подозрительны. Расположенные в каталогах пользователей файлы, владельцем которых является суперпользователь и у которых установлен бит SETUID, представляют собой потенциальную проблему безопасности, так как такие файлы при запуске любым пользователем приобретают полномочия суперпользователя. Список технически возможных, но необычных ситуаций, о которых программа должна сообщать, можно продолжать довольно долго.

До сих пор мы обсуждали проблему защиты пользователя от сбоев. Некоторые файловые системы также пытаются защитить пользователя от самого себя. Если пользователь собирается ввести команду

```
rm *.o
```

чтобы удалить все файлы с расширением .o (созданные компилятором объектные файлы), но случайно вместо этого набьет строку

```
rm * .o
```

(обратите внимание на пробел после звездочки), программа *rm* удалит все файлы в текущем каталоге, после чего сообщит, что не может найти файл *.o*. В системе MS-DOS и некоторых других системах при удалении файла устанавливается всего лишь один бит в каталоге или в i-узле, отмечая, что файл удален. Блоки диска не возвращаются[1] в список свободных блоков до тех пор, пока они не понадобятся. Таким образом, если пользователь быстро обнаружит ошибку, он сможет восстановить удаленные файлы. В системе Windows удаленные файлы обычно помещаются в мусорную корзину, откуда их можно при необходимости извлечь. При этом свободное пространство на диске не увеличивается до тех пор, пока корзина не будет очищена.

[1] В MS-DOS первый символ имени удаленного файла заменяется символом 0xE5, а блоки, занимаемые файлом, освобождаются. Создаваемый после этого новый файл может занять эти блоки и запись в каталоге. Но сразу после удаления файл может быть восстановлен по сохранившемуся (кроме первого символа имени) описателю в каталоге. — *Примеч. перев.*

Производительность файловой системы

Доступ к диску производится значительно медленнее, чем к оперативной памяти. Чтение слова из памяти может занять около 10 нс. Чтение с жесткого диска может выполняться со скоростью 10 Мбайт/с, что в сорок раз медленнее, но к этому следует добавить 5–10 мс на поиск нужного цилиндра и задержку вращения диска. Если требуется прочитать или записать всего одно слово, то оперативная память оказывается примерно в миллион раз быстрее жесткого диска. Поэтому во многих файловых системах применяются различные методы оптимизации, увеличивающие производительность. В данном разделе мы рассмотрим три из них.

Кэширование

Для минимизации количества обращений к диску применяется **блочный кэш** или **буферный кэш**. (Термин «кэш» происходит от французского слова *cacher*, что значит «скрывать»). В данном контексте кэшем называется набор блоков, логически принадлежащих диску, но хранящихся в оперативной памяти по соображениям производительности.

Существуют различные алгоритмы управления кэшем. Обычная практика заключается в перехвате всех запросов чтения к диску и проверке наличия требующихся блоков в кэше. Если блок присутствует в кэше, то запрос чтения блока может быть удовлетворен без обращения к диску. В противном случае блок сначала считывается с диска в кэш, а оттуда копируется по нужному адресу памяти. Последующие обращения к тому же блоку могут удовлетворяться из кэша.

Работа кэша показана на рис. 6.24. Поскольку в кэше хранится большое количество (часто тысячи) блоков, требуется некий быстрый способ определения наличия или отсутствия блока в кэше. Обычно для этого используется хэширование номера устройства и дискового адреса (номера блока) и поиск результата в хэш-таблице. Все блоки с одинаковыми хэш-кодами сцепляются вместе в связный список.

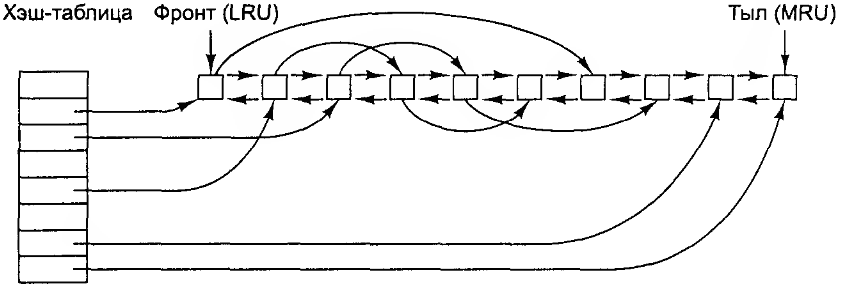


Рис. 6.24. Структура данных блочного кэша

Когда требуется загрузить блок в заполненный до предела кэш, какой-либо другой блок должен быть удален из кэша (и записан на диск, если он был модифицирован в кэше). Эта ситуация очень похожа на страничную организацию памяти, и к ней применимы все обычные алгоритмы замены, описанные в главе 3, такие как FIFO (First in First Out — первым прибыл — первым обслужен), «вторая попытка» и LRU (Least Recently Used — с наиболее давним использованием). Одно

приятное отличие кэширования от страничной организации памяти состоит в том, что обращения к кэшу производятся относительно нечасто, что позволяет хранить все блоки в точном LRU-порядке со связными списками.

На рис. 2.24 мы видим, что в дополнение к цепям, начинающимся в хэш-таблице, используется также и двунаправленный список, в котором содержатся номера всех блоков в порядке их использования. При этом самый старый блок помещается в начало списка, а самый новый блок — в его конец. При обращении к блоку блок может перемещаться со своей текущей позиции в конец двунаправленного списка. Таким образом, может поддерживаться точный LRU-порядок.

К сожалению, здесь есть одна загвоздка. Теперь, когда мы можем реализовать точное выполнение алгоритма LRU, оказывается, что алгоритм LRU является нежелательным. Вызвано это тем, что буквальное применение алгоритма LRU снижает надежность файловой системы и угрожает ее непротиворечивости (обсуждавшейся в предыдущем разделе). Если в кэш считывается и модифицируется критический блок, например блок i-узла, но не записывается тут же на диск, то компьютерный сбой может привести к тому, что файловая система окажется в противоречивом состоянии. Если блок i-узла поместить в конец цепочки LRU, то может пройти довольно много времени, прежде чем этот блок попадет в ее начало и будет записан на диск.

Более того, к некоторым блокам, таким как блоки i-узлов, программы редко обращаются дважды в течение короткого интервала времени. Исходя из этих соображений, мы приходим к модифицированной схеме LRU, принимая во внимание два следующих фактора:

1. Насколько велика вероятность того, что данный блок скоро снова понадобится?
2. Важен ли данный блок для непротиворечивости файловой системы?

Для ответа на каждый из этих вопросов блоки можно разделить на такие категории, как блоки i-узлов, косвенные блоки, блоки каталогов, блоки, полные данных, и блоки, частично заполненные данными. Блоки, которые, вероятно, не потребуются снова в ближайшее время, помещаются в начало списка LRU, чтобы занимаемые ими буферы могли вскоре освободиться. Блоки, вероятность повторного использования которых в ближайшее время высока (например, записываемые блоки, частично заполненные данными), помещаются в конец списка LRU, что позволяет им оставаться в кэше более долгое время.

Второй вопрос не связан с первым. Если блок представляет важность для непротиворечивости файловой системы (обычно это все блоки, кроме блоков данных) и такой блок модифицируется, то его следует немедленно сохранить на диске независимо от его положения в списке LRU. Быстро записывая критические блоки, мы значительно снижаем вероятность того, что сбой компьютера повредит файловую систему. Пользователь вряд ли будет рад потере одного из своих файлов из-за сбоя компьютера. Еще более он огорчится, если при этом испорченной окажется вся файловая система.

Даже при принятии всех перечисленных выше мер предосторожности по поддержанию в рабочем состоянии файловой системы слишком долгое хранение в кэше блоков с данными является нежелательным. Представьте себе пользовате-

ля, работающего на персональном компьютере над написанием книги. Даже если наш писатель периодически велит текстовому редактору сохранять редактируемый файл на диске, есть большая вероятность, что все блоки останутся в кэше. Если произойдет сбой, структура файловой системы не пострадает, но работа целого дня работы будет потеряна.

Эта ситуация случается не слишком часто, если только с очень невезучими пользователями. Для решения данной проблемы обычно применяется два метода. В системе UNIX есть системный вызов sync, принуждающий сохранение всех модифицированных блоков кэша на диске. При загрузке операционной системы запускается фоновая задача, обычно называющаяся *update*, вся работа которой заключается в периодическом (обычно через каждые 30 с) обращении к системному вызову sync. В результате при любом сбое будет потеряно не более 30 с работы.

В системе MS-DOS используется другой подход, состоящий в том, что каждый модифицированный блок записывается на диск сразу же. Кэш, в котором все модифицированные блоки немедленно записываются на диск, называются **сквозным кэшем** или **кэшем со сквозной записью**. При использовании сквозного кэша количество обращений ввода-вывода к диску больше, чем при применении обычного кэша. Чтобы лучше понять разницу в этих двух подходах, представьте себе программу, записывающую блок размером в 1 Кбайт по одному символу. Система UNIX будет собирать все символы в кэше и записывать этот блок на диск каждые 30 с или когда блок будет удален из кэша. Система MS-DOS будет обращаться к диску при каждом записываемом символе. Конечно, большинством программ применяется внутренняя буферизация, поэтому обычно они обращаются к системному вызову write не с одним символом, а с целыми строками или большими единицами данных.

Результатом различия стратегий кэширования оказывается тот факт, что простое удаление (гибкого) диска из системы UNIX, без выполнения системного вызова sync, почти всегда приведет к потере данных и часто также к повреждению файловой системы. В MS-DOS такой проблемы не возникает. Такое различие в стратегиях связано с тем, что система UNIX разрабатывалась в среде, в которой все диски были жесткими и постоянными, тогда как система MS-DOS изначально предназначалась для работы с гибкими дисками. Когда жесткие диски стали нормой, более эффективный метод, применяющийся в UNIX, стал нормой и теперь также используется в Windows для жестких дисков.

Опережающее чтение блока

Второй метод увеличения производительности файловой системы состоит в попытке получить блоки диска в кэш прежде, чем они потребуются. В частности, многие файлы считываются последовательно. Когда файловая система получает запрос на чтение блока k файла, она выполняет его, но после этого сразу проверяет, есть ли в кэше блок $k + 1$. Если этого блока в кэше нет, файловая система читает его в надежде, что к тому моменту, когда он понадобится, этот блок уже будет считан в кэш. В крайнем случае, он уже будет на пути туда.

Конечно, такая стратегия работает только для тех файлов, которые считываются последовательно. Если обращения к блокам файла производятся в случайном порядке, опережающее чтение не помогает. В самом деле, не хотелось бы обременять

диск считыванием ненужных блоков и удалением потенциально полезных блоков из кэша (возможно, тем самым еще более обременяя диск необходимостью записывать эти блоки на диск, если они модифицированы). Чтобы определить, следует ли использовать опережающее чтение блоков, файловая система может вести учет доступа к блокам каждого открытого файла. Например, для каждого открытого файла один бит может означать «режим последовательного доступа» или «режим произвольного доступа». Вначале каждому открываемому файлу в соответствии с принципом презумпции невиновности назначается режим последовательного доступа. Однако при перемещении указателя в файле этот бит сбрасывается. Если к этому файлу опять будут обращаться с запросами последовательного чтения, бит будет установлен снова. Таким образом, файловая система может строить догадки о том, следует ли ей выполнять операции опережающего чтения или нет. Если она и будет ошибаться время от времени, то ничего страшного не произойдет, просто будет потрачен впустую некоторый процент пропускной способности диска.

Снижение времени перемещения блока головок

Кэширование и опережающее чтение являются не единственными способами увеличения производительности системы. Другой важный метод состоит в уменьшении затрат времени на перемещение блока головок. Достигается это помещением блоков, к которым высока вероятность доступа в течение короткого интервала времени, близко друг к другу, желательно на одном цилиндре. Когда записывается выходной файл, файловая система должна зарезервировать место для чтения таких блоков за одну операцию. Если свободные блоки учитываются в битовом массиве, а весь битовый массив помещается в оперативной памяти, то довольно легко выбрать свободный блок как можно ближе к предыдущему блоку. В случае когда свободные блоки хранятся в списке, часть которого в оперативной памяти, а часть на диске, сделать это значительно труднее.

Однако даже при использовании списка свободных блоков может быть выполнена определенная кластеризация блоков. Хитрость заключается в том, чтобы учитывать место на диске не в блоках, а в группах последовательных блоков. Если сектор состоит из 512 байт, система может использовать блоки размером в 1 Кбайт (2 сектора), но выделять пространство на диске в единицах по 2 блока (4 сектора). Это не то же самое, что использование 2-килобайтных дисковых блоков, так как кэш по-прежнему будет использовать килобайтные блоки и дисковые операции чтения и записи будут по-прежнему работать с килобайтными блоками. Однако при последовательном чтении файла количество операций поиска цилиндра уменьшится вдвое, что значительно увеличит производительность. Вариация этой же темы состоит в попытке системы учесть позицию блока в цилиндре.

Еще один фактор, снижающий производительность файловых систем, связан с тем, что при использовании i-узлов или чего-либо эквивалентного им, особенно при чтении коротких файлов, требуется два обращения к диску вместо одного: одно для i-узла и одно для блока данных. Обычное размещение i-узлов на диске показано на рис. 6.25, *а*. Здесь все i-узлы располагаются в начале диска, так что среднее расстояние между i-узлом и его блоками будет составлять около половины количества цилиндров, то есть при доступе практически к каждому файлу потребуются значительные перемещения блока головок.

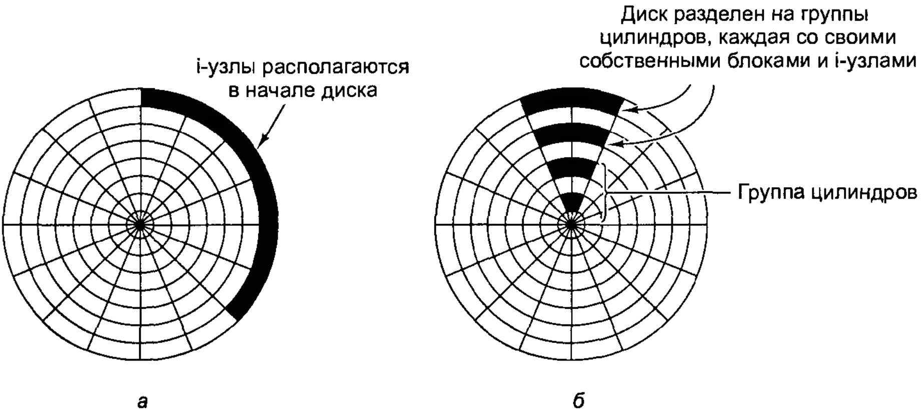


Рис. 6.25. I-узлы, размещенные в начале диска (*а*); диск, разделенный на группы цилиндров, каждая со своими собственными блоками и i-узлами (*б*)

Один из способов увеличения производительности состоит в помещении i-узлов в середину диска, уменьшая, таким образом, среднее расстояние перемещения блоков головок в два раза. Другая идея, показанная на рис. 2.25, *б*, заключается в разбиении диска на группы цилиндров, каждая со своими i-узлами, блоками и списком свободных блоков [230]. Когда создается новый файл, может быть выбран любой i-узел, но предпринимается попытка найти блок в той же группе цилиндров, что и i-узел. Если эта попытка заканчивается неудачей, используется блок в соседней группе цилиндров.

Файловые системы с журнальной структурой LFS

Изменения в технологии оказывают влияние на современные файловые системы. В частности, центральные процессоры становятся все быстрее, емкость дисков увеличивается, цена их снижается (но скорость их увеличивается не столь значительно), а размеры оперативной памяти растут экспоненциально. Единственным параметром, не растущим столь стремительно, является время поиска цилиндра диска. В результате во многих файловых системах появляется узкое место. В университете Беркли были проведены исследования, направленные на снижение остроты этой проблемы при помощи создания совершенно новой файловой системы LFS (Log-structured File System — файловая система с журнальной структурой). В этом разделе мы кратко опишем, как работает система LFS. Дополнительные сведения можно узнать в [280].

В основе системы LFS лежит идея того, что по мере ускорения центральных процессоров и увеличения размеров оперативной памяти кэширование дисков становится все выгоднее. Поэтому становится возможным удовлетворить весьма существенную часть всех дисковых запросов прямо из кэша файловой системы без обращения к диску. Из этого следует, что в будущем большинство обращений к диску будут составлять обращения записи, поэтому алгоритм опережающего чтения, применявшийся в некоторых файловых системах, уже не дает большого выигрыша производительности.

Ситуация усложняется тем, что в большинстве файловых систем операции записи выполняются очень маленькими блоками данных. Такие операции записи оказываются крайне неэффективными, поскольку самой записи, занимающей 50 мкс, часто предшествует поиск цилиндра в течение 10 мс и 4-миллисекундная задержка вращения. При таких параметрах эффективность диска падает до 1 %.

Чтобы понять, из чего складываются все эти мелкие операции записи, рассмотрим создание файла в операционной системе UNIX. Для записи в файл необходимо произвести операции записи в i-узел каталога, блок каталога, i-узел файла и, наконец, блок самого файла. В принципе эти операции записи могут быть отложены на некоторое время, но при этом могут возникнуть серьезные проблемы с непротиворечивостью файловой системы в случае сбоя компьютера прежде, чем будет выполнена запись на диск. По этой причине i-узлы обычно записываются на диск без промедления.

Учитывая все вышесказанное, разработчики файловой системы LFS решили реализовать файловую систему UNIX таким образом, чтобы достичь максимума эффективности использования диска, несмотря на рабочую нагрузку, состоящую из большого количества случайных мелких операций записи. Замысел состоит в использовании диска как журнала. Периодически, когда возникает необходимость, все буферизированные в памяти блоки, которые должны быть записаны, собираются вместе в единый сегмент, и он записывается на диск одним непрерывным куском в конец журнала. Записываемый сегмент может содержать i-узлы, блоки каталогов и блоки данных, перемешанные друг с другом. В начале каждого сегмента создается оглавление сегмента. Если довести средний размер сегмента до 1 Мбайт, то можно использовать почти всю пропускную способность диска.

При такой организации i-узлы существуют и имеют ту же структуру, что и в UNIX, но теперь они, вместо того чтобы располагаться в фиксированной позиции на диске, перемешаны по всему журналу. Тем не менее, когда программа находит i-узел, определение расположения блоков выполняется обычным способом. Конечно, теперь обнаружить i-узел намного сложнее, так как его адрес не определяется по его номеру, как это было в UNIX. Чтобы можно было найти i-узел, создается массив i-узлов, индексированный по i-номерам. Элемент *i* массива указывает на i-узел i на диске. Массив хранится на диске, но также содержится и в кэше, так что наиболее часто используемые части этого массива постоянно находятся в оперативной памяти.

Таким образом, все операции записи буферизируются в памяти, и периодически данные из буфера записываются на диск в виде единых сегментов в конец журнала. Чтобы открыть файл, используется массив, позволяющий обнаружить i-узел в файле. Как только i-узел обнаружен, могут быть определены номера блоков файла. Все эти блоки также располагаются в сегментах где-то в журнале.

Если бы диски были бесконечного размера, на этом все бы и заканчивалось. Однако существующие диски имеют ограниченный размер, поэтому рано или поздно журнал займет весь диск. К счастью, многие сегменты могут содержать уже ненужные блоки, например, если файл был перезаписан, его i-узел будет указывать на новые блоки, но старые блоки будут все также занимать место в записанных ранее сегментах.

Для решения проблемы повторного использования блоков в старых сегментах у файловой системы LFS есть **чистящий поток**, занимающийся постоянным скани-

рованием журнала с целью сделать его более компактным. Он начинает с того, что считывает содержание самого первого сегмента журнала, чтобы определить, какие i-узлы и файлы находятся в нем. Затем он смотрит в текущий массив i-узлов, проверяя, являются ли i-узлы все еще текущими и используются ли все еще блоки файлов. Если нет, то эта информация отбрасывается, а все еще использующиеся i-узлы и блоки считываются в память, чтобы записать их в следующий сегмент. Исходный сегмент помечается как свободный, поэтому журнал может использовать его для новых данных. Таким образом, чистильщик двигается по журналу, удаляя старые сегменты с диска и помещая всю имеющую ценность информацию в память для перезаписи в следующий сегмент. В результате диск представляет собой большой кольцевой буфер, в котором пишущий поток добавляет новые сегменты с одного конца, а чистящий процесс удаляет старые сегменты с другого.

Учет расположения блоков здесь весьма нетривиален, поскольку, когда блок файла записывается в новый сегмент, i-узел файла (где-то в журнале) должен быть найден, обновлен и помещен в буфер для записи в следующий сегмент. При этом массив i-узлов также должен быть обновлен, чтобы элемент массива указывал на новую копию. Тем не менее администрирование такой системы вполне возможно, а увеличение производительности показывает, что все эти сложности были не напрасны. Приведенные в цитированной выше статье результаты измерений показали, что файловая система с журнальной структурой LFS превосходит систему UNIX при множестве небольших записях на порядок, а при чтении и больших записях обладает сходной или лучшей производительностью.

Примеры файловых систем

В следующих разделах мы обсудим несколько примеров файловых систем, от довольно простых до крайне сложных. Поскольку современные файловые системы UNIX и Windows 2000 будут описываться в отдельных главах этой книги (главы 10 и 11), мы не станем обсуждать их здесь. Вместо них мы рассмотрим их предшественников.

Файловые системы CD-ROM

В качестве первого примера рассмотрим файловую систему, применяемую на CD-ROM. Эти системы являются совсем простыми, поскольку они создавались для носителей, запись на которые могла производиться только один раз. Кроме всего прочего, в этих файловых системах отсутствует учет свободных блоков, так как файлы на CD-ROM не могут быть удалены или добавлены после того, как диск уже изготовлен. Ниже мы рассмотрим основные типы файловых систем для CD-ROM и два варианта расширения этих систем.

Файловая система ISO 9660

В 1988 году был принят Международный стандарт **ISO 9660**, описывающий файловые системы для CD-ROM. Практически каждый продающийся сегодня CD-ROM соответствует этому стандарту, иногда согласуясь с его расширениями, которые

будут обсуждаться ниже. Одно из назначений этого стандарта заключалось в том, чтобы любой CD-ROM мог быть прочитан на любом компьютере, независимо от используемых байтового порядка и операционной системы. Как следствие, на файловую систему были наложены определенные ограничения, которые должны были позволить читать эти диски даже самым слабым из использовавшихся тогда операционных систем (таким как MS-DOS).

У CD-ROM нет концентрических цилиндров, как у магнитных дисков. Вместо этого они содержат непрерывную спираль, на которой последовательно размещены все биты (хотя поиск поперек спирали также возможен). Биты вдоль спирали разделены на логические блоки (также называемые логическими секторами) по 2352 байт. Некоторые из этих байтов используются для преамбул, коррекции ошибок и других накладных расходов. Полезная нагрузка в каждом блоке составляет 2048 байт. Аудиодиски содержат специальные разделительные участки между композициями, а также специальные заголовки и концевики для каждой фонограммы, не используемые в CD-ROM, содержащих другие данные. Часто позиция блока в спирали указывается в минутах и секундах. Она может быть преобразована в линейный номер блока, так как каждая секунда содержит 75 блоков.

Стандарт ISO 9660 также поддерживает наборы размером до $2^{16}-1$ CD-ROM. Сами CD-ROM также могут быть разделены на отдельные логические тома (разделы). Однако ниже будет обсуждаться стандарт ISO 9660 для одного CD-ROM, не разделенного на тома.

Каждый CD-ROM начинается с 16 блоков, чья функция не определяется стандартом ISO 9660. Производитель CD-ROM может использовать эту область для размещения загрузчика операционной системы или для другой цели. Следом располагается один блок, содержащий **основной описатель тома**, в котором хранится некоторая общая информация о CD-ROM. Среди данных, содержащихся в этом блоке, идентификатор системы (32 байт), идентификатор тома (32 байт) идентификатор издателя (128 байт), и идентификатор лица, подготовившего данные (128 байт). Производитель диска может заполнить эти поля произвольным образом, с условием, что он будет использовать только символы верхнего регистра, цифры и очень ограниченное количество знаков препинания, чтобы гарантировать совместимость с различными платформами.

Основной описатель тома также содержит имена трех файлов, в которых могут храниться краткий обзор, уведомление об авторских правах и библиографическая информация соответственно. Кроме того, в этом блоке также содержатся определенные ключевые числа, включающие размер логического блока (как правило, 2048, однако в определенных случаях могут использоваться блоки большего размера, например 4096, 8192 и других степеней двух), количество блоков на CD-ROM, а также дата создания и дата окончания срока службы диска. Наконец, основной описатель тома также содержит описатель корневого каталога, что позволяет найти этот каталог на CD-ROM (то есть определить номер блока, содержащего начало каталога). Начиная с этого каталога, можно определить местонахождение всей остальной файловой системы.

Помимо основного описателя тома, CD-ROM-диск может содержать дополнительный описатель тома. В нем хранится информация, сходная с хранящейся в основном описателе, но мы не станем рассматривать его в этой книге.

Корневой каталог и все остальные каталоги могут содержать переменное количество записей, в последней из которых установлен специальный бит, помечающий эту запись как последнюю. Сами каталоговые записи также могут иметь переменную длину. Каждая запись содержит от 10 до 12 полей, некоторые из них содержат текст формата ASCII, а другие являются числовыми двоичными полями. Двоичные поля кодируются дважды, один раз в формате, используемом в процессорах типа Pentium (сначала младшие байты, затем старшие), и один раз в формате, используемом в процессоре SPARC (сначала старшие байты, затем младшие). Следовательно, 16-разрядное число занимает 4 байт, а 32-разрядное число 8 байт. Такое избыточное кодирование было использовано при разработке стандарта, чтобы никого не обидеть. Если бы стандарт учитывал только один из способов хранения двоичного числа, тогда сотрудники компаний, в которых применяется другой способ, посчитали бы, что их отнесли к гражданам второго сорта и не приняли бы стандарт. Таким образом, эмоциональное содержание CD-ROM может быть точно измерено в килобайтах потерянного пространства.

Формат каталоговой записи стандарта ISO 9660 показан на рис. 6.26. Поскольку каталоговые записи могут быть переменной длины, первое поле записи представляет собой байт, содержащий длину записи. Во избежание любых двусмысленностей стандартом определено, что старший бит этого байта располагается слева.

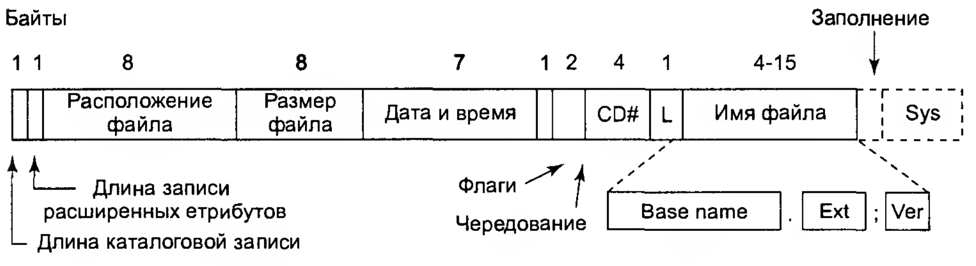


Рис. 6.26. Каталоговая запись стандарта ISO 9660

Записи каталогов могут иметь расширенные атрибуты. Если для каталоговой записи используется это свойство, тогда второй байт содержит длину записи расширенных атрибутов.

Следом располагается номер начального блока файла. Файлы хранятся на диске в виде непрерывных последовательностей блоков, так что размещение файла на диске однозначно определяется начальным блоком и размером, значение которого содержится в следующем поле.

В следующем поле хранятся дата и время записи CD-ROM[1]. Значения года, месяца, дня, часа, минуты, секунды и временной зоны хранятся в отдельных байтах. Годы отсчитываются от 1900, что означает, что CD-ROM будут страдать от проблемы 2156 года, так как следом за 2155 годом для них наступит 1900 год. Возникновение этой проблемы можно было отложить на 88 лет, приняв за точку отсчета 1988 (год принятия стандарта). Если бы это было сделано, проблему можно было бы отложить до 2244 года.

[1] Вернее, дата и время создания файла. — *Примеч. перев.*

Поле *Flags* (флаги) содержит несколько различных управляющих битов, один из которых позволяет скрывать запись при отображении каталога (свойство, взятое из MS-DOS), другой разрешает использование расширенных атрибутов, а третий помечает последнюю запись в каталоге. Мы не станем рассматривать здесь остальные биты этого поля. Следующее поле описывает особенности чередования частей файла на диске. Это свойство не используется в простейшей версии стандарта ISO 9660, поэтому оно не будет обсуждаться в данной книге.

Еще одно поле указывает местоположение файла на CD-ROM. Стандарт допускает возможность расположения файла на другом CD-ROM набора. Таким образом, можно создать на одном CD-ROM главный каталог, содержащий все файлы всех остальных CD-ROM набора.

Поле, отмеченное на рис. 6.26 символом *L*, содержит длину имени файла в байтах. За ним следует само имя файла, состоящее из базового имени (*base name* на рисунке), точки, расширения, точки с запятой и версии файла в двоичном формате (один или два байта). В базовом имени и расширении могут использоваться прописные символы, цифры от 0 до 9 и символ подчеркивания. Все остальные символы запрещены, чтобы гарантировать, что каждый компьютер сможет работать со всеми файлами на диске. Базовое имя может быть длиной до восьми символов; расширение — до трех символов. Такой выбор был продиктован необходимостью совместимости с системой MS-DOS. Имя файла может встречаться несколько раз, но с различными номерами версий.

Последние два поля не всегда присутствуют. Поле *Padding* (заполнение) используется для выравнивания размера каталоговой записи до четного количества байтов, чтобы выровнять записи в каталоге по 2-байтовым границам. Если требуется выравнивание, используется нулевой байт. Наконец, функция и размер последнего поля *System use* (Sys на рисунке) никак не определяются стандартом. В стандарте указывается лишь, что это поле должно состоять из четного числа байтов. В различных операционных системах это поле используется различным образом. Например, Macintosh хранит в этом поле флаги Finder.

Все записи каталога, кроме первых двух, располагаются в алфавитном порядке. Первая запись представляет собой описатель самого каталога. Вторая запись является ссылкой на родительский каталог. В этом смысле эти записи аналогичны каталоговым записям «.» и «..» в UNIX.

Количество каталоговых записей не ограничено. Однако существует ограничение глубины вложенности каталогов. Максимальная глубина вложенности каталогов равна восьми.

Стандартом ISO 9660 определены так называемые три уровня. На уровне 1 применяются самые жесткие ограничения. Имена файлов ограничиваются уже описанной выше схемой 8 + 3, а имена каталогов могут состоять из восьми символов и не могут иметь расширений. Кроме того, уровень 1 требует, чтобы все файлы были непрерывными. Использование этого уровня обеспечивает совместимость CD-ROM с самым широким спектром систем.

Уровень 2 ослабляет ограничение на длину имени. Он позволяет файлам и каталогам иметь имена до 31 символа, но из того же набора символов.

На уровне 3 используются те же ограничения имен, что и на уровне 2, но ослабляется жесткость требования непрерывности файлов. На этом уровне файл может состоять из нескольких разделов, каждый из которых представляет собой непрерывную последовательность блоков. Одна и та же последовательность блоков может несколько раз встречаться в одном файле и даже входить в несколько различных файлов. Такая организация файловой системы позволяет экономить место на диске.

Рок-Ридж расширения

Как было показано, стандарт ISO 9660 содержит много различных ограничений. Вскоре после выхода этого стандарта пользователи из UNIX-сообщества начали работу над его расширением, чтобы файловая система UNIX могла быть представлена на CD-ROM. Эти расширения получили название Рок-Ридж (Rock Ridge) по городу из фильма Джина Уайлдера *Blazing Saddles* (Огненные седла), вероятно, потому, что этот фильм нравился одному из членов комитета.

Расширение использует поле *System use*, чтобы CD-ROM формата Рок-Ридж мог читаться на любом компьютере. Все остальные поля соответствуют требованиям стандарта ISO 9660. Система, не знакомая с расширениями Рок-Ридж, просто игнорирует их и видит нормальный CD-ROM.

Расширения содержат следующие поля:

1. PX — Атрибуты POSIX.
2. PN — Старший и младший номера устройств.
3. SL — Символьная связь.
4. NM — Альтернативное имя.
5. CL — Расположение дочернего узла.
6. PL — Расположение дочернего узла.
7. RE — Перераспределение.
8. TF — Временные штампы.

Поле *PX* содержит стандартные биты разрешений *rwxrwxrwx* системы UNIX для владельца, группы и всех остальных. Оно также содержит остальные биты слова состояния, такие как SETUID, SETGID и т. п.

Чтобы необработанные устройства могли быть представлены на CD-ROM, вводится поле *PN*. Оно содержит старший и младший номера устройств, ассоциированных с файлом. Таким образом, содержимое каталога */dev* может быть записано на CD-ROM и позднее правильно воссоздано на другой системе.

Поле *SL* используется для символьных связей. Оно позволяет файлу из одной файловой системы ссылаться на файл из другой файловой системы.

Вероятно, наиболее важным является поле *NM*. С его помощью можно указать для файла второе имя. Этого имени не касаются ограничения стандарта ISO 9660, что позволяет указывать произвольные имена файлов системы UNIX на CD-ROM.

Следующие три поля используются вместе, чтобы обойти ограничения стандарта ISO 9660 на глубину вложенности каталогов. С их помощью можно указать, куда в дереве иерархии должен быть перемещен тот или иной каталог.

Наконец, поле *TF* содержит три временных штампа, включаемые в каждый i-узел системы UNIX, а именно: время создания файла, последнего изменения файла и последнего доступа к файлу. Все вместе эти расширения позволяют скопировать файловую систему UNIX на CD-ROM, а затем корректно восстановить ее на другой машине.

Расширения Joliet

Сообщество UNIX было не единственной группой, которой требовалось расширение стандарта ISO 9660. Корпорация Microsoft также пришла к выводу, что стандарт ISO 9660 содержит слишком много ограничений (хотя большинство ограничений было вызвано в первую очередь требованием совместимости с файловой системой MS-DOS фирмы Microsoft). Поэтому корпорация Microsoft разработала некоторые расширения, названные **Joliet**. Они должны были позволить копировать на CD-ROM-диск и восстанавливать с него файловую систему Windows подобно тому, как расширения Рок-Ридж позволяли работать с файловой системой UNIX. Теоретически все программы, работающие в операционной системе Windows и использующие CD-ROM, включая программы записи на CD-R, поддерживают расширение Joliet.

Основными расширениями, содержащимися в Joliet, являются:

1. Длинные имена файлов.
2. Набор символов Unicode.
3. Глубина вложенности каталогов, превышающая восемь уровней.
4. Имена каталогов с расширениями.

Первое расширение позволяет использовать имена файлов длиной до 64 символов. Второе расширение разрешает использовать для имен файлов символы Unicode. Это расширение важно для программного обеспечения, предназначенного для распространения в странах, в которых не используется латинский алфавит, таких как Япония, Израиль или Греция[1]. Поскольку символы Unicode занимают два байта, максимальное имя файла в расширении Joliet занимает 128 байт.

Как и Рок-Ридж, расширение Joliet устраняет ограничение на глубину вложенности каталогов. Каталоги могут вкладываться друг в друга на любую требуемую глубину. Наконец, у имен каталогов могут быть расширения. Неясно, почему было включено такое расширение стандарта, поскольку каталоги в файловой системе Windows практически никогда не используют расширений, но, возможно, однажды они потребуются.

Файловая система CP/M

Первые персональные компьютеры (называвшиеся тогда мини-компьютерами) появились в начале 80-х годов. В одной популярной модели персонального компьютера использовался 8-разрядный центральный процессор Intel 8080. У этого компьютера было 4 Кбайт ОЗУ и один 8-дюймовый накопитель для сменных гибких дисков емкостью 180 Кбайт. В более поздних версиях этой машины применялся

[1] И Россия. – *Примеч. перев.*

несколько более мощный (хотя все равно 8-разрядный) процессор Zilog Z80. У него было до 64 Кбайт ОЗУ и огромный 720-килобайтный гибкий диск в качестве основного устройства хранения данных. Несмотря на невысокое быстродействие и небольшое количество памяти, почти на всех этих машинах работала удивительно мощная дисковая операционная система **CP/M** (Control Program for Microcomputers — управляющая программа для микрокомпьютеров) [130]. Эта операционная система была доминирующей в свою эпоху, так же как MS-DOS и позднее Windows стали лидерами в мире IBM PC. Двадцать лет спустя она исчезла, не оставив следа (если не считать малочисленной группы твердокаменных сторонников), и это наводит на мысль о том, что системы, лидирующие сегодня в мире (например, Windows), могут оказаться никому неизвестными к тому времени, когда сегодняшние карапузы станут студентами колледжей.

Стоит взглянуть на операционную систему CP/M по нескольким причинам. Во-первых, исторически это была очень важная система, ставшая прямым предшественником системы MS-DOS. Во-вторых, разработчики сегодняшних операционных систем и систем будущего, полагающие, что компьютеру требуется 32 Мбайт памяти, только чтобы загрузить в нее операционную систему, могут поучиться простоте системы, которой вполне хватало 16 Кбайт ОЗУ. В-третьих, в ближайшие десятилетия широкое применение получат встроенные системы. В связи с ограничениями в цене, размерах, весе и потребляемой мощности операционные системы, используемые, например, в часах, видео- и фотокамерах, радиоприемниках и сотовых телефонах, обязательно должны быть компактными, подобно операционной системе CP/M. Конечно, у этих систем не будет 8-дюймовых гибких дисков, но они вполне могут пользоваться электронными дисками во флэш-ОЗУ, на которых несложно организовать файловую систему, подобную CP/M.

Распределение памяти в системе CP/M показано на рис. 6.27. Наверху оперативной памяти (в ОЗУ) располагается базовая система ввода-вывода BIOS, содержащая базовую библиотеку — 17 вызовов ввода-вывода, используемых системой CP/M (в данном разделе мы рассмотрим CP/M 2.2, являвшуюся стандартной версией, когда CP/M была на вершине популярности). Эти системные вызовы предназначены для чтения и записи с гибкого диска, ввода с клавиатуры и вывода на экран.

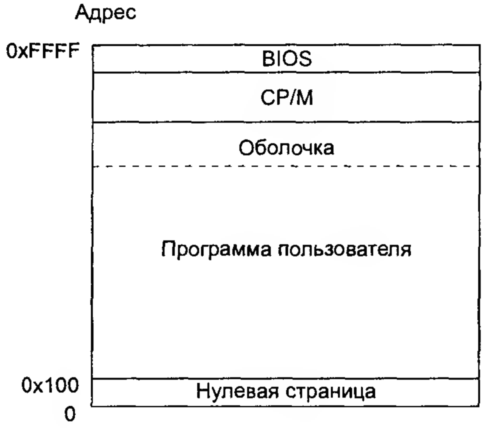


Рис. 6.27. Распределение памяти в системе CP/M

Сразу под BIOS располагается сама операционная система. Для версии CP/M 2.2 ее размер составляет 3584 байт. Невероятно, но факт: вся операционная система занимала менее 4 Кбайт. Под операционной системой размещается оболочка (обработчик командных строк), съедающая еще 2 Кбайт. Остальная память используется для программ пользователя, кроме нижних 256 байт, зарезервированных для векторов аппаратных прерываний, некоторых переменных и буфера для текущей командной строки, в котором к ней могут получить доступ программы пользователя.

Причина, по которой система BIOS отделена от самой операционной системы CP/M (хотя обе системы располагаются в ОЗУ), заключается в переносимости. Операционная система CP/M взаимодействует с аппаратурой только с помощью обращений к BIOS. Для переноса системы CP/M на новую машину нужно всего лишь перенести туда BIOS. Когда это сделано, сама CP/M может быть установлена без изменений.

В файловой системе CP/M всего один каталог, содержащий записи фиксированного размера (32 байт). Размер каталога, фиксированный для данной реализации, может отличаться в других реализациях системы CP/M. В этом каталоге перечисляются все файлы системы. После загрузки система считывает каталог и рассчитывает битовый массив занятых и свободных блоков. Этот битовый массив, размер которого для 180-килобайтного диска составляет всего 23 байта, постоянно хранится в оперативной памяти. После завершения работы операционной системы он не сохраняется на диске. Благодаря такому подходу исчезает необходимость в проверке[1] непротиворечивости файловой системы на диске (вроде той, что выполняет программа *fsck* в UNIX) и сохраняется на диске один блок (в процентном отношении это эквивалентно сохранению 90 Мбайт на современном 16-гигабайтном диске).

Когда пользователь набирает команду, оболочка сначала копирует ее в буфер в нижние 256 байт памяти. Затем она ищет вызываемую программу и загружает ее в память по адресу 256 (над вектором прерываний), после чего передает управление по этому адресу. Программа начинает работу. Она обнаруживает свои параметры в буфере командной строки. Программе разрешается использовать память, занимаемую оболочкой, если ей нужно много памяти. Закончив работу, программа выполняет системный вызов CP/M, сообщая операционной системе, что следует перезагрузить оболочку (если занимаемая ею память использовалась программой) и запустить ее. В двух словах, вот, собственно, и весь рассказ об операционной системе CP/M.

Помимо загрузки программ, система CP/M предоставляет программам пользователя 38 системных вызовов, большей частью относящихся к файловой службе. Наиболее важными из них являются системные вызовы чтения из файла и записи в файл. Прежде чем файл может быть прочитан, его следует открыть. Когда CP/M получает системный вызов open, она должна прочитать свой единственный каталог и найти в нем требуемый файл. Для экономии драгоценной памяти этот каталог не хранится в ОЗУ постоянно. Когда CP/M обнаруживает описатель файла, она сразу же получает содержащиеся прямо в нем номера дисковых блоков файла, а также все остальные атрибуты. Формат каталоговой записи показан на рис. 6.28.

[1] В результате сбоя файлы могут пересечься, то есть один и тот же блок может оказаться сразу в двух файлах. Так что проверка все равно нужна. Например, при загрузке операционной системы. — *Примеч. перев.*

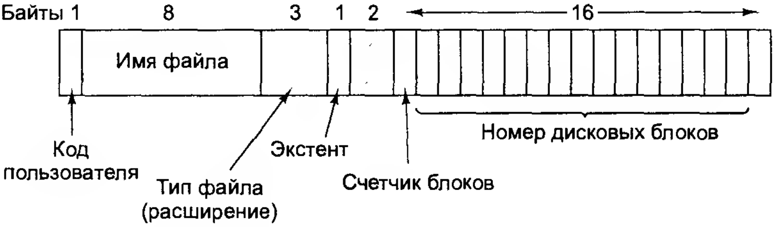


Рис. 6.28. Формат каталоговой записи в системе CP/M

Значение полей на рис. 6.28 следующее. Поле *User code* (код пользователя) указывает владельца файла. Хотя в каждый конкретный момент времени в системе CP/M может находиться лишь один пользователь, системой поддерживается поочередная работа нескольких пользователей. При поиске имени файла файловая система проверяет только записи, принадлежащие текущему зарегистрировавшемуся пользователю. В результате каждый пользователь получает свой виртуальный каталог без необходимости управлять несколькими каталогами.

В следующих двух полях содержатся имя и расширение файла. Размер базового имени может быть до восьми символов. Также может использоваться (необязательное) расширение файла длиной до трех символов. Формат имени файла из 8 + 3 символов верхнего регистра был позднее заимствован в MS-DOS.

Поле *Block count* (счетчик блоков) содержит размер файла, измеренный в единицах по 128 байт (так как ввод-вывод выполняется в 128-байтовых[1] физических секторах). Последний килобайтный блок файла может быть заполнен не до конца, поэтому у системы нет способа определить точный размер файла. Конец файла может быть при необходимости указан пользователям с помощью специального маркера. Последние 16 полей содержат сами номера дисковых блоков, занимаемых файлом. Размер каждого блока 1 Кбайт, поэтому максимальный размер файла равен 16 Кбайт. Обратите внимание, что физический ввод-вывод выполняется в 128-байтовых секторах и размер файла хранится в 128-байтовых секторах, но файлам выделяются блоки размером по 1 Кбайт (сразу по 8 секторов), чтобы не увеличивать размер каталоговой записи.

Однако разработчик системы CP/M понимал, что некоторые файлы, даже на 180-килобайтном гибком диске, могут превзойти размер 16 Кбайт, поэтому для обхода 16-килобайтного ограничения был придуман следующий трюк. Для файла размером от 16 до 32 Кбайт используется не одна каталоговая запись, а две. В первой записи содержатся номера первых 16 блоков диска; во второй записи — следующие 16 блоков. При превышении файлом 32 Кбайт требуется третья каталоговая запись и т. д. Порядковый номер каталоговой записи хранится в поле *Extent* (экстент), благодаря которому операционная система может определить, какие 16 Кбайт находятся в начале файла, какие идут следом и т. д.

После того как системный вызов open выполнен, адреса всех дисковых блоков становятся известны, поэтому реализация системного вызова read не представля-

[1] Малый размер физических блоков (128 байт) значительно замедляет операции ввода-вывода, тогда как большой размер кластеров (1 Кбайт) сильно снижает эффективность использования дискового пространства. Поэтому лучше было бы сделать наоборот, то есть физические блоки по 1 Кбайт, а логические — по 128 или 256 байт. — *Примеч. перев.*

ет сложности. Системный вызов write также прост. Для этого требуется всего лишь выделение файлу нового свободного блока из битового массива, хранящегося в оперативной памяти, и запись блока. Последовательные блоки файла не располагаются последовательно на диске, так как центральный процессор Intel 8080 не успевает обработать прерывание и начать чтение следующего блока. Вместо этого используется чередование блоков, позволяющее считывать несколько блоков за один оборот диска.

Система CP/M, очевидно, не является последним словом в области современных файловых систем, но она отличается своей простотой, скоростью и может быть реализована компетентным программистом менее чем за неделю. Для многих встроенных приложений, возможно, большего и не требуется.

Файловая система MS-DOS

В первом приближении файловая система MS-DOS представляет собой увеличенную и улучшенную версию CP/M. Она работает только на платформах с центральным процессором Intel, не поддерживает многозадачности и работает лишь в реальном режиме IBM PC (изначально этот режим был единственным режимом). Оболочка содержит больше возможностей, чем в CP/M, системных вызовов тоже больше в MS-DOS, но основной функцией операционной системы все также остается загрузка программ, управление экраном, обработка ввода с клавиатуры и управление файловой системой. Именно последняя функция и будет интересовать нас в данной главе.

Файловая система MS-DOS во многом напоминает файловую систему CP/M, включая использование имен файлов, состоящих из 8 + 3 символов верхнего регистра. В первой версии системы (MS-DOS 1.0) был даже всего один каталог, как и в CP/M. Однако, начиная с MS-DOS 2.0, функциональность файловой системы значительно расширилась. Самым серьезным улучшением явился переход на иерархическую файловую систему, в которой каталоги могли вкладываться друг в друга на произвольную глубину. Это означало, что корневой каталог (размер которого по-прежнему был ограничен) мог содержать подкаталоги, которые, в свою очередь, также могли содержать подкаталоги и т. д. до бесконечности. Связи, принятые в UNIX, не допускались, поэтому файловая система представляла собой дерево, начинавшееся в корневом каталоге.

Различные прикладные программы довольно часто начинают с того, что создают в корневом каталоге подкаталог, в который складывают все свои файлы, что позволяет программам избежать конфликта. Так как сами каталоги хранятся в MS-DOS как файлы, нет никакого ограничения на число каталогов или файлов на диске. Однако в отличие от CP/M, в MS-DOS нет концепции различных пользователей. Соответственно, любой вошедший в систему пользователь получает доступ сразу ко всем файлам.

Чтобы прочитать файл, программа, работающая в системе MS-DOS, должна сначала сделать системный вызов open, чтобы получить дескриптор файла. Системному вызову open в качестве одного из входных аргументов следует указать путь к файлу, который может быть как абсолютным, так и относительным (относительно текущего каталога). Файловая система открывает каталоги, перечисленные

в пути, один за другим, пока не обнаруживает последний каталог, который считывается в оперативную память. Затем в считанном каталоге ищется описатель файла, который требуется открыть.

Хотя каталоги в файловой системе MS-DOS переменного размера, используемые каталоговые записи, как и в CP/M, имеют фиксированный размер 32 байт. Формат описателя файла системы MS-DOS показан на рис. 6.29. В нем содержится имя файла, его атрибуты, дата и время создания, номер начального блока и точный размер файла. Имена файлов короче 8 + 3 символов выравниваются по левому краю полей и дополняются пробелами, каждое поле отдельно. Поле *Attributes* (атрибуты) представляет собой новое поле, содержащее биты, указывающие, что для файла разрешено только чтение, что файл должен быть заархивирован, что файл является системным или скрытым. Запись в файл, для которого разрешено только чтение, не разрешается. Таким образом осуществляется защита файлов от случайной записи или удаления. Бит *archived* (архивный) не устанавливается и не проверяется операционной системой MS-DOS. Он зарезервирован в описателе для архивирующих программ уровня пользователя, сбрасывающих этот бит при создании резервной копии файла, в то время как программы, модифицирующие файл, должны устанавливать этот бит. Таким образом архивирующая программа может определить, какие файлы подлежат архивации. Бит *hidden* (скрытый файл) позволяет избежать отображения файла в перечне файлов каталога. Основное его назначение заключается в том, чтобы скрыть от неопытных пользователей файлы, назначение которых им неизвестно. Наконец, бит *system* (системный) также скрывает файлы и защищает их от случайного удаления командой *del*. Этот бит установлен у основных компонентов системы MS-DOS.

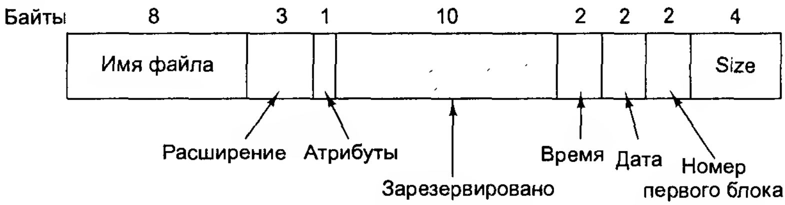


Рис. 6.29. Формат каталоговой записи в системе MS-DOS

Каталоговая запись также содержит дату и время создания или последнего изменения файла. Время хранится с точностью ±2 с[1], так как для него отведено 2-байтовое поле, способное содержать всего 65 536 уникальных значений, а в сутках 86 400 с. Поле времени разбивается на подполя секунд (5 бит), минут (6 бит) и часов (5 бит). 16-разрядное поле даты также разбивается на три подполя: день (5 бит), месяц (4 бит) и год — 1980 (7 бит). При 7 двоичных разрядах для хранения года и 1980 в качестве точки отсчета максимальное значение года, которое можно выразить таким способом, — это 2107-й. Таким образом, файловая система MS-DOS имеет встроенную проблему 2108 года. Чтобы избежать катастрофы, пользователи системы MS-DOS должны начать готовиться к 2108 году как можно раньше. Если бы в MS-DOS использовался единый 32-разрядный счетчик секунд, точность

[1] Точнее, ±1 с, то есть с шагом в 2 с. — *Примеч. перев.*

представления времени была бы в два раза выше, а катастрофу можно было бы отложить до 2116 года.

В отличие от файловой системы CP/M, не хранящей точного размера файла, система MS-DOS хранит точный размер. Поскольку для размера файла используется 32-разрядное число, в теории файлы могут быть размером до 4 Гбайт. Однако другие пределы (описываемые ниже) ограничивают максимальный размер файла размером 2 Гбайт или меньше. Удивительно большая часть описателя файла (10 байт) не используется.

Другое отличие системы MS-DOS от CP/M состоит в том, что дисковые адреса файлов не хранятся прямо в их описателях, возможно, поскольку разработчики системы понимали, что большие жесткие диски (обычные в то время на мини-компьютерах) однажды появятся в мире MS-DOS. Вместо этого файловая система MS-DOS хранит номера блоков файла в специальной таблице размещения файлов, в оперативной памяти. В каталоговой записи хранится номер первого блока файла. Этот номер используется в качестве индекса для 64 К[1] элементов FAT-таблицы, хранящейся в оперативной памяти. Все блоки файла могут быть найдены, если проследовать по цепочке элементов таблицы. Принцип действия FAT-таблицы проиллюстрирован на рис. 6.11.

В зависимости от количества блоков на диске в системе MS-DOS применяется три версии файловой системы FAT: FAT-12, FAT-16 и FAT-32. В действительности FAT-32 является неверным названием, так как используются только 28 младших битов дискового адреса. Ее следовало бы назвать FAT-28, но степени двух звучат гораздо приятнее.

Во всех файловых системах FAT размер блока диска в байтах может быть установлен равным некоторому числу, кратному 512 (возможно, различному для каждого раздела диска), с наборами разрешенных размеров блоков (называемых корпорацией Microsoft **размерами кластеров**), различными для каждого варианта FAT. В первой версии системы MS-DOS использовалась FAT-12 с 512-байтовыми блоками, что позволяло создавать дисковые разделы размером до $2^{12} \times 512$ байт (на самом деле только 4086×512 байт, так как 10 дисковых адресов использовались как специальные маркеры — конец файла, дефектный блок и т. д.). При этом максимальный размер дискового раздела мог составлять 2 Мбайт, а в оперативной памяти FAT-таблица занимала 4096 элементов по два байта каждый. Кроме того, обработка 12-разрядных адресов была довольно медленной.

Такая система неплохо работала на гибких дисках, но с появлением жестких дисков появились проблемы. Корпорация Microsoft попыталась решить проблему, разрешив использовать дисковые блоки (кластеры) размером 1, 2 и 4 Кбайт. Это позволило сохранить структуру и размер таблицы FAT-12 и увеличить размер дискового раздела до 16 Мбайт.

Так как система MS-DOS поддерживала до четырех дисковых разделов на диске, новая файловая система FAT-12 могла работать с дисками емкостью до 64 Мбайт. Для поддержки винчестеров большего размера нужно было предпринимать что-то еще. В результате была разработана файловая система FAT-16, с 16-разрядными

---

[1] Справедливо только для FAT-16. Например, для FAT-12, до сих пор применяющейся для гибких дисков, таблица содержит $2^{12} = 4096$ элементов. Для FAT-32 размер таблицы во много раз превышает 64 К. — *Примеч. перев.*

дисковыми указателями. Дополнительно было разрешено использовать кластеры размеров 8, 16 и 32 Кбайт. (32 768 — это максимальное число, представляющее собой степень двух, которое может быть представлено 16 двоичными разрядами.) Теперь таблица FAT-16 постоянно занимала 128 Кбайт оперативной памяти, но с ростом размеров памяти компьютеров она получила широкое применение и быстро вытеснила файловую систему FAT-12. Максимальный размер дискового раздела, поддерживаемый системой FAT-16, равен 2 Гбайт (64 К элементов по 32 Кбайт каждый), а максимальный размер диска — 8 Гбайт, то есть четыре раздела по 2 Гбайт каждый.

Для хранения деловой переписки такое ограничение проблем не вызывает, однако при работе с цифровым видео в стандарте DV один 2-гигабайтный файл вмещает всего лишь немногим более 9 мин видеофильма. Таким образом, на весь диск из четырех разделов может поместиться около 38 мин видео, независимо от размеров самого диска. Это ограничение также означает, что в режиме on-line редактировать можно не более 19 мин фильма, так как на диске требуется одновременно хранить и входные и выходные файлы.

С выходом второй версии операционной системы Windows 95 была представлена файловая система FAT-32 со своими 28-разрядными адресами. При этом версия системы MS-DOS, лежащая в основе Windows 95, была адаптирована для поддержки FAT-32. Теоретически в этой системе разделы могли быть по $2^{28} \times 2^{15}$ байт, но фактически размер разделов ограничен 2 Тбайт (2048 Гбайт), так как внутренне система учитывает размеры разделов в 512-байтовых секторах с помощью 32-разрядных чисел, а $2^{32} \times 2^{9}$ байт равно 2 Тбайт. Максимальный размер раздела для всех трех типов FAT и различных размеров кластеров показан в табл. 6.3. Пустым элементам таблицы соответствуют запрещенные комбинации.

Таблица 6.3. Максимальный размер раздела для различных размеров кластеров

Размер кластера, Кбайт	FAT-12, Мбайт	FAT-16, Мбайт	FAT-32, Тбайт
0,5	2		
1	4		
2	8	128	
4	16	256	1
8		512	2
16		1024	2
32		2048	2

Помимо поддержки дисков большего размера, файловая система FAT-32 обладает двумя другими преимуществами перед системой FAT-16. Во-первых, 8-гигабайтный диск, использующий FAT-32, может состоять из всего одного раздела. При использовании FAT-16 он должен был содержать четыре раздела, что представлялось пользователям системы Windows как логические устройства C:, D:, E: и F:. Какой файл на каком устройстве располагать, решать пользователю.

Другое преимущество FAT-32 перед FAT-16 заключается в том, что для дискового раздела заданного размера могут использоваться блоки меньшего размера. Например, для 2-гигабайтного дискового раздела система FAT-16 должна пользо-

ваться 32-килобайтными блоками, в противном случае при наличии всего 64 К доступных дисковых адресов она не смогла бы покрыть весь раздел. В то же время система FAT-32 для такого же дискового раздела может использовать, например, блоки размером 4 Кбайт. Преимущество блоков меньшего размера заключается в том, что длина большинства файлов менее 32 Кбайт. При размере блока в 32 Кбайт даже 10-байтовый файл будет занимать на диске 32 Кбайт. Если средний размер файлов, скажем, равен 8 Кбайт, тогда при использовании 32-килобайтных блоков около 3/4 дискового пространства будет теряться, то есть эффективность использования диска будет низкой. При 8-килобайтных файлах и 4-килобайтных блоках потерь дискового пространства не будет, но платой за это будет то, что для хранения таблицы FAT потребуется значительно больше оперативной памяти. При 4-килобайтных блоках 2-гигабайтный раздел будет состоять из 512 К блоков, поэтому таблица FAT должна состоять из 512 К элементов (занимая 2 Мбайт ОЗУ).

Файловая система MS-DOS использует FAT для учета свободных блоков. Любой незанятый блок помечается специальным кодом. Когда системе MS-DOS требуется новый блок на диске, она ищет этот код в таблице FAT. Таким образом, битовый массив или список свободных блоков не нужны.

Файловая система Windows 98

Первая версия операционной системы Windows 95 использовала файловую систему MS-DOS, с именами файлов из 8 + 3 символов и системами FAT-12 и FAT-16. Начиная со второй версии системы Windows 95, были разрешены более длинные имена файлов. Кроме того, была представлена система FAT-32, в первую очередь для поддержки дисков размером более 8 Гбайт и дисковых разделов, больших, чем 2 Гбайт. Та же файловая система была использована в системе Windows 98. Ниже будут описаны эти особенности файловой системы Windows 98, также применяющейся в системе Windows Me.

Поскольку длинные имена файлов производят на пользователей более сильное впечатление, чем структура FAT, рассмотрим их в первую очередь. Для того чтобы позволить пользователям работать с длинными именами файлов, можно было просто разработать новую структуру каталога. Недостаток такого подхода состоит в том, что если бы корпорация Microsoft сделала это, пользователи, до сих пор переходящие с Windows 3 на Windows 95 или Windows 98, не смогли бы иметь доступ ко всем своим файлам одновременно из обеих систем. В корпорации Microsoft было принято политическое решение о том, что файлы, созданные в системе Windows 98, должны быть также доступны из Windows 3 (для машин с двойной загрузкой). В результате была разработана система поддержки длинных имен, обладавшая обратной совместимостью со старой системой имен 8 + 3, применявшейся в MS-DOS. Поскольку такие требования обратной совместимости нередки в компьютерной промышленности, стоит детально ознакомиться с тем, как корпорация Microsoft выполнила эту задачу.

Итак, каталоговая структура системы Windows 98 должна была обладать обратной совместимостью со структурой каталогов MS-DOS. Как уже было показано, эта структура представляет собой просто список 32-байтовых описателей (рис. 6.30). Такой формат был заимствован у файловой системы CP/M (написанной для про-

цессора Intel 8080), что демонстрирует, насколько долго структуры (устаревшие) могут существовать в компьютерном мире.

Однако в 32-байтовом описателе файла оставались незадействованными 10 байт (см. рис. 6.29), что и было использовано. Это изменение не имеет никакого отношения к длинным именам, но используется в Windows 98, поэтому стоит обратить на него внимание.

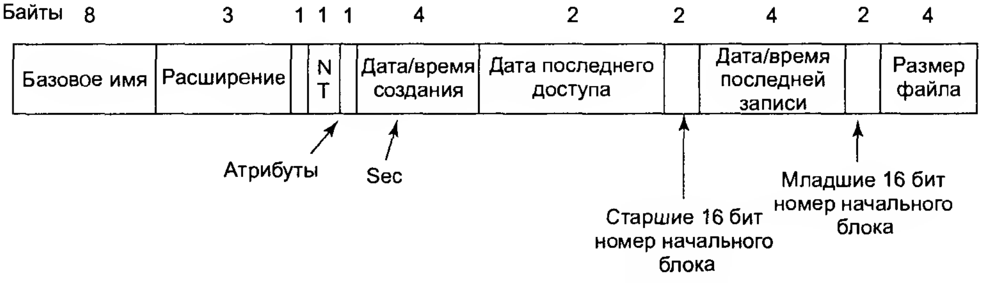


Рис. 6.30. Формат каталоговой записи в системе MS-DOS

Изменение каталоговой записи состоит в добавлении пяти новых полей на место неиспользовавшихся 10 байт. Поле *NT* предназначено для совместимости с Windows NT и обеспечивает отображение имени файла в правильном регистре. Поле *Sec* решает проблему невозможности хранения времени суток в 16-битовом поле с точностью до секунды. Восемь дополнительных разрядов позволяют хранить поле *Creation time* (время создания) с точностью до 10 мс. Еще одно новое поле *Last access* (дата последнего доступа) хранит дату (но не время) последнего доступа к файлу. Наконец, переход на файловую систему FAT-32 означает, что номера блоков теперь 32-разрядные, поэтому для хранения старших разрядов номера начального блока файла требуются дополнительные 16 бит.

Теперь мы подходим к сердцу файловой системы Windows 98: способу представления длинных имен файлов, обладающему обратной совместимостью с MS-DOS. Выбранное решение заключается в назначении каждому файлу двух имен: (потенциально) длинного имени файла (в формате Unicode, для совместимости с Windows NT) и имени формата 8 + 3 для совместимости с MS-DOS. Доступ к файлам может быть получен по любому имени. Когда создается файл, имя которого не удовлетворяет правилам MS-DOS (8 символов для имени и 3 символа для расширения, ограниченный набор символов, отсутствие пробелов и т. д.), Windows 98 создает дополнительное имя формата MS-DOS в соответствии с определенным алгоритмом. Берутся первые шесть символов имени, преобразуются при необходимости в верхний регистр ASCII, после чего к ним добавляется суффикс ~1. Если такое имя уже есть, то используется суффикс ~2 и т. д. Кроме того, удаляются пробелы и лишние точки, а определенные символы преобразуются в символы подчеркивания. Например, имя файла *The time has come the walrus said*[1] получает имя формата MS-DOS *THETIM~1*. Если впоследствии создается файл с именем *The time has come the rabbit said*, ему назначается имя *THETIM~2* и т. д.

[1] «Тюлень сказал: „Пришла пора...“». Строка из книги Льюиса Кэрролла «Алиса в Зазеркалье». — *Примеч. перев.*

Имя формата MS-DOS хранится в каталоге прямо в описателе, показанном на рис. 6.30. Если у файла есть также длинное имя, оно хранится в одной или нескольких каталоговых записях, предшествующих описателю файла с именем в формате MS-DOS. Каждая такая запись содержит до 13 символов формата Unicode. Элементы имени хранятся в обратном порядке, начинаясь сразу перед описателем файла в формате MS-DOS и последующими фрагментами перед ним. Формат каждого фрагмента длинного имени показан на рис. 6.31.

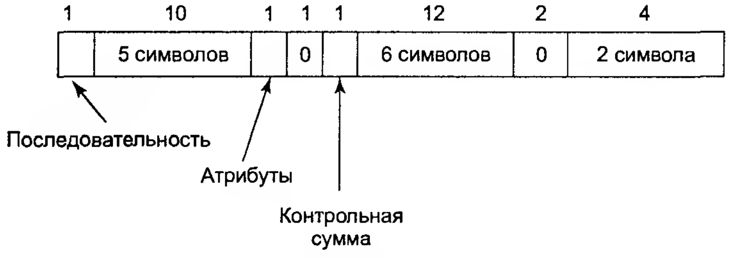


Рис. 6.31. Формат каталоговой записи с фрагментом длинного имени файла в Windows 98

Возникает очевидный вопрос: «Как файловая система Windows 98 отличает каталоговые записи, содержащие имя файла в формате MS-DOS, от фрагментов длинных имен?» Ответ содержится в поле *Attributes* (атрибуты). Для фрагмента длинного имени это поле содержит значение 0x0F, что соответствует невозможной комбинации атрибутов для описателя файла в MS-DOS. Старые программы, написанные для работы в MS-DOS, читая каталог, просто игнорируют такие описатели как неверные. Порядок фрагментов имени учитывается в первом байте каталоговой записи. Последний фрагмент имени отмечается добавлением к порядковому номеру числа 64. Поскольку для порядкового номера используется всего 6 бит, теоретически максимальная длина имени файла может составить 63 × 13 = 819 символов. На практике она ограничена 260 символами по историческим причинам.

Каждый фрагмент длинного имени содержит поле *Checksum* (контрольная сумма) во избежание следующей проблемы. Сначала программа, работающая в системе Windows 98, создает файл с длинным именем. Затем компьютер перезагружается в MS-DOS или Windows 3. После этого старая программа, удаляя файл, удаляет из каталога имя формата MS-DOS, но оставляет в нем предшествующее ему длинное имя (так как ей ничего не известно о длинных именах). Наконец, какая-то программа создает новый файл, используя освободившееся место в каталоге. К этому моменту мы имеем верную последовательность фрагментов длинного имени, предшествующую описателю файла формата MS-DOS, который не имеет к ней никакого отношения. Поле *Checksum* позволяет системе Windows 98 обнаружить такую ситуацию. Конечно, поскольку для поля *Checksum* используется всего один байт, есть один шанс из 256, что Windows 98 не заметит подмены.

Рассмотрим пример использования длинного имени файла на рис. 6.32. В данном случае файл назван *The quick brown fox jumps over the lazy dog*[1]. Поскольку в этом имени 42 символа, то оно определенно квалифицируется как длинное. Имя формата MS-DOS, созданное из него, выглядит как *THEQUI~1* и хранится в последней записи.

[1] Незаконченная любимая фраза программистов, содержащая все символы английского алфавита. Полностью фраза выглядит так: «*The quick brown fox jumps over the lazy dog's tail*». — *Примеч. перев.*

68	d o g	A	0	C K					0	
3	o v e	A	0	C K	t h e l a				0	z y
2	w n f o	A	0	C K	x j u m p				0	s
1	T h e q	A	0	C K	u i c k b				0	r o
T	H E Q U L ~ 1	A	N T	S	Creation time	Last acc	Upp	Last write	Low	Size

Байты

Рис. 6.32. Пример хранения длинного имени файла в Windows 98

Каталоговая структура обладает некоторой избыточностью, позволяющей обнаруживать проблемы, вызванные вмешательством старых программ, написанных для работы в Windows 3. Порядковый номер в начале каждого фрагмента длинного имени не так уж и нужен, так как бит 0x40 помечает первый фрагмент, но он обеспечивает дополнительную избыточность. Кроме того, поле *Low* на рис. 6.32 (младшая половина номера начального кластера) равно 0 во всех фрагментах, кроме последнего, что также позволяет избежать неверной интерпретации этих каталоговых записей старыми программами и разрушения файловой системы. Байт *NT* используется системой Windows NT и игнорируется системой Windows 98. Байт *A* содержит атрибуты файла.

Реализация файловой системы FAT-32 концептуально близка к реализации файловой системы FAT-16. Однако вместо массива из 65 536 элементов в ней используется столько, сколько нужно, чтобы покрыть весь раздел диска. Если диск содержит миллион блоков, то и таблица будет состоять из миллиона элементов. Для экономии памяти система Windows 98 не хранит их все сразу в памяти, а использует окно, накладываемое на таблицу.

Файловая система UNIX V7

Даже в ранних версиях системы UNIX применялась довольно сложная многопользовательская файловая система, так как в основе этой системы лежала операционная система MULTICS. Ниже будет рассмотрена файловая система V7, разработанная для компьютера PDP-11, сделавшего систему UNIX знаменитой. Современные версии будут обсуждаться в главе 10.

Файловая система представляет собой дерево, начинающееся в корневом каталоге, с добавлением связей, формирующих направленный ациклический граф. Имена файлов могут содержать до 14 символов, включающих в себя любые символы ASCII, кроме косой черты (использовавшейся в качестве разделителя компонентов пути) и символа NUL (использовавшегося для дополнения имен короче 14 символов). Символ NUL обозначается байтом 0.

Каталог UNIX содержит по одной записи для каждого файла этого каталога. Каждая каталоговая запись максимально проста, так как в системе UNIX используется схема i-узлов (см. рис. 6.12). Каталоговая запись состоит всего из двух полей: имени файла (14 байт) и номера i-узла для этого файла (2 байт), как пока-

зано на рис. 6.33. Эти параметры ограничивают количество файлов в файловой системе числом 64 К.

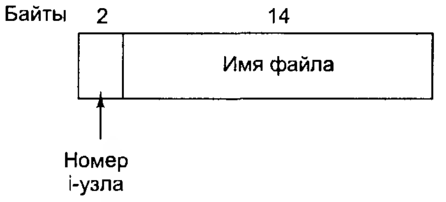


Рис. 6.33. Каталоговая запись файловой системы UNIX V7

Как и i-узлы на рис. 6.12, i-узлы системы UNIX содержат некоторые атрибуты. К этим атрибутам относятся такие параметры, как размер файла, три указателя времени (создания, последнего доступа и последнего изменения), идентификатор владельца, номер группы, информация о защите и счетчик каталоговых записей, указывающих на этот i-узел. Последнее поле необходимо для связей. При добавлении новой связи к i-узлу счетчик в i-узле увеличивается на единицу. При удалении связи счетчик в i-узле уменьшается на единицу. Когда значение счетчика достигает нуля, i-узел освобождается, а блоки диска, которые занимал файл, возвращаются в список свободных блоков.

Для учета дисковых блоков файла используется обобщение схемы, показанной на рис. 6.12, позволяющее работать с очень большими файлами. Первые 10 дисковых адресов хранятся в самом i-узле. Таким образом, для небольших файлов вся необходимая информация содержится прямо в i-узле, считываемом с диска при открытии файла. Для файлов большего размера один из адресов в i-узле представляет собой адрес блока диска, называемого **одинарным косвенным блоком**. Этот блок содержит дополнительные дисковые адреса. Если и этого недостаточно, используется другой адрес в i-узле, называемый **двойным косвенным блоком**, и содержащий адрес блока, в котором хранятся адреса однократных косвенных блоков. Если и этого мало, используется **тройной косвенный блок**. Полная схема показана на рис. 6.34.

При открытии файла файловая система по имени файла находит его блоки на диске. Рассмотрим на примере открытие файла */usr/ast/mbox*. В качестве примера будем использовать файловую систему UNIX, хотя основы алгоритма одинаковы для всех иерархических каталоговых систем. Сначала файловая система открывает корневой каталог. В системе UNIX его i-узел располагается в фиксированном месте диска. По этому i-узлу система определяет положение корневого каталога, который может находиться в любом месте диска, в данном примере — в блоке 1.

Затем файловая система считывает корневой каталог и ищет в нем первый компонент пути, *usr*, чтобы определить номер i-узла файла */usr*. Определить местоположение i-узла несложно, так как для каждого из них предусмотрено фиксированное место на диске. По этому i-узлу файловая система находит каталог */usr* и находит в нем следующий компонент, *ast*. Найдя описатель *ast*, файловая система получает i-узел для каталога */usr/ast*. По этому i-узлу она получает доступ к самому каталогу, в котором ищет файл *mbox*. При этом i-узел файла *mbox* считывается в память и остается там, пока файл не будет закрыт. Процесс поиска проиллюстрирован на рис. 6.35.

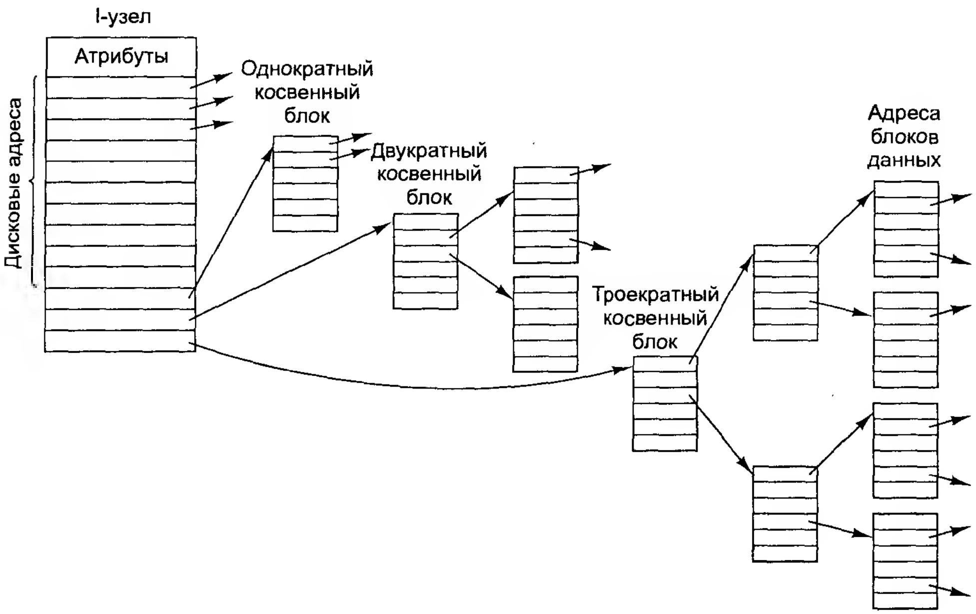


Рис. 6.34. I-узел файловой системы UNIX V7

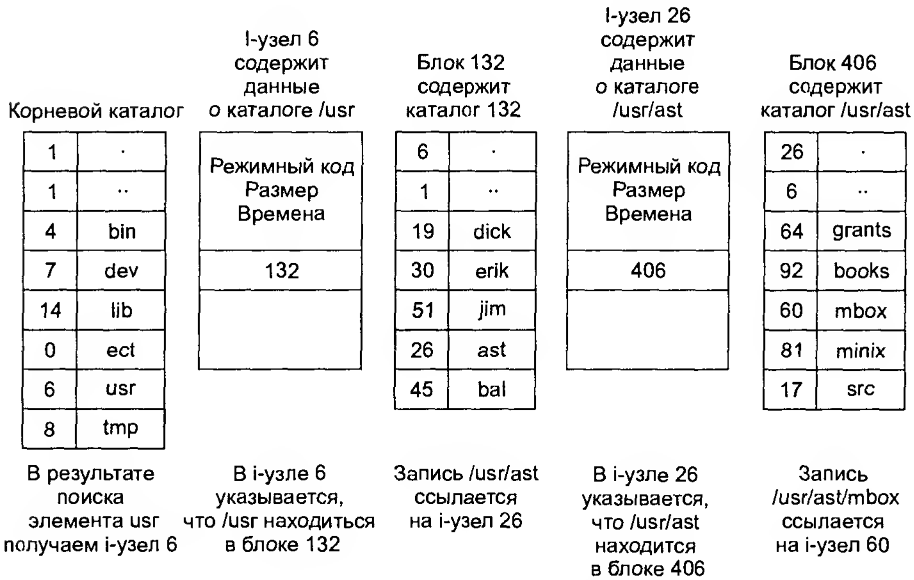


Рис. 6.35. Этапы поиска файла */usr/ast/mbox*

Относительные пути файлов обрабатываются так же, как и абсолютные, с той разницей, что алгоритм начинает работу не с корневого, а с рабочего каталога. В каждом каталоге есть элементы «.» и «..», помещаемые в каталог в момент его создания. Элемент «.» содержит номер i-узла текущего каталога, а элемент «..» —

номер i-узла родительского каталога. Таким образом, процедура, ищущая файл *../dick/prog.c*, просто находит «..» в рабочем каталоге, разыскивает в нем номер i-узла родительского каталога, в котором ищет описатель каталога *dick*. Для обработки этих имен не требуется специального механизма. Имена рабочего и родительского каталогов представляют собой обычные ASCII-строки, не отличающиеся от любых других имен.

Исследования в области файловых систем

Файловым системам всегда уделялось больше внимания исследователей, чем другим частям операционной системы, и такая ситуация сохраняется до сих пор. Некоторые подобные исследования касаются структуры файловой системы. Последнее время популярными являются файловые системы с журнальной структурой LFS [226, 356]. Логический диск разбивает файловую систему на два уровня: файловую систему и дисковую систему [85]. Построение файловой системы из наращиваемых уровней также составляет тему исследований [152].

Расширяемые файловые системы аналогичны расширяемым ядрам, рассматривавшимся в главе 1. Они позволяют добавлять к файловым системам новые возможности без необходимости пересоздавать их с нуля [174, 180].

Другой популярной темой исследований являются измерения состава и использования файловой системы. Исследователи измеряют такие факторы, как распределение размеров файлов, время жизни файлов, частоту обращений к файлам, соотношение частот операций чтения и записи и многие другие параметры [98, 128, 281, 346].

Другие исследователи занимались вопросами производительности файловой системы и способами улучшения опережающего чтения, кэширования, снижения количества операций копирования и т. д. Как правило, эти исследователи производят измерения, определяют, где находятся узкие места, удаляют, по меньшей мере, одно из них, после чего снова производят измерения на улучшенной системе, чтобы проверить свои результаты [52, 256, 260].

Архивация и восстановление файловой системы представляет собой тему, о которой многие пользователи редко задумываются до тех пор, пока не случится что-либо страшное. В этой области также появляются новые идеи оптимизации [62, 94, 160]. С этой темой связан вопрос о том, что делать, когда пользователь удаляет файл — удалить его или скрыть? Например, файловая система Elephant никогда ничего не забывает [290, 291].

Резюме

При взгляде снаружи файловая система представляет собой набор файлов и каталогов плюс операции с ними. Файлы могут считываться и записываться, каталоги могут создаваться и удаляться. Файлы могут перемещаться из одного каталога в другой. В большинстве современных файловых систем поддерживается иерархическая каталоговая система, в которой каталоги могут содержать подкаталоги, которые, в свою очередь, также могут иметь подподкаталоги и т. д. До бесконечности.

При взгляде изнутри файловая система выглядит совсем по-другому. Разработчикам файловых систем приходится заботиться о том, как файлам выделяется место на диске, и о том, как система следит за тем, какой блок какому файлу принадлежит. Различные варианты реализации файлов включают в себя непрерывные файлы, связные списки, таблицы размещения файлов и i-узлы. В различных системах используются различные каталоговые структуры. Атрибуты файла могут храниться прямо в каталоге или в другом месте (например, в i-узле). Учет дискового пространства может осуществляться с помощью списков свободных блоков или битовых массивов. Надежность файловых систем может быть увеличена при помощи создания инкрементных резервных копий, а также с помощью программы, способной исправлять поврежденные файловые системы. Производительность файловых систем также является важным вопросом. Она может быть увеличена различными способами, включая кэширование, опережающее чтение и размещение блоков файла рядом друг с другом. Файловые системы с журнальной структурой тоже увеличивают производительность, выполняя операции записи большими блоками данных.

Файловые системы описаны на примере ISO 9660, CP/M, MS-DOS, Windows 98 и UNIX. Они во многом отличаются друг от друга, в частности способом учета принадлежности блоков файлам, каталоговой структурой и управлением свободным дисковым пространством.

Вопросы

1. Создайте пять различных путей к файлу */etc/passwd*. *Подсказка*: используйте элементы каталога «.» и «..».
2. В системе Windows, когда пользователь дважды щелкает мышью на файле, отображаемом в программе Windows Explorer, запускается программа, которой в качестве параметра передается имя этого файла. Перечислите два различных способа указания операционной системе, какую программу следует запускать при двойном щелчке мышью.
3. В ранних версиях системы UNIX исполняемые файлы (файлы *a.out*) начинались со специфического «магического» числа, не являвшегося случайным. Эти файлы начинались с заголовка, за которым следовали сегменты с текстом программы и данными. Как вы думаете, почему для исполняемых файлов было выбрано особенное «магическое» число, тогда как для файлов других типов использовались более или менее случайные «магические» числа в качестве первого слова?
4. В табл. 6.2 один из атрибутов представляет собой длину записи. Зачем этот атрибут нужен операционной системе?
5. Является ли необходимым системный вызов open в системе UNIX? Какими будут последствия его отсутствия?
6. В системах, поддерживающих последовательный доступ, всегда имеется операция для перемотки файлов. Нужна ли такая операция в системах, поддерживающих файлы произвольного доступа?

7. В некоторых операционных системах предоставляется системный вызов rename, позволяющий сменить имя файла. Есть ли разница между использованием этого системного вызова и копирования файла с новым именем с последующим удалением старого файла?
8. Некоторые системы позволяют отображать часть файла на память. Какие ограничения должны быть наложены на такую систему? Как реализуется такое частичное отображение файла на память?
9. Простая операционная система поддерживает только один каталог, но позволяет хранить в нем произвольное количество файлов с именами произвольной длины. Можно ли на такой системе симулировать иерархическую файловую систему? Как?
10. В системах UNIX и Windows произвольный доступ к файлу осуществляется при помощи специального системного вызова, перемещающего указатель текущей позиции в файле на новое место. Предложите альтернативный метод реализации произвольного доступа без использования этого системного вызова.
11. Рассмотрите дерево каталогов на рис. 6.7. Если */usr/jim* является рабочим каталогом, как будет выглядеть абсолютный путь для файла с относительным путем *../ast/x*?
12. Как указывалось в тексте, работа с непрерывными файлами приводит к фрагментации диска, так как теряется некоторая часть дискового пространства в последних блоках файлов, чья длина не кратна целому числу блоков. Является эта фрагментация внутренней или внешней? Приведите аналогию с вопросом, обсуждавшимся в предыдущей главе.
13. Один из способов использовать непрерывные файлы на диске и не страдать от дыр состоит в уплотнении диска при каждом удаленном файле. Поскольку все файлы являются непрерывными, для копирования файла требуется определенное время на поиск цилиндра и вращение диска при считывании файла, после которого происходит перенос данных на полной скорости. При записи файла на диск требуются аналогичные операции. При времени поиска цилиндра, равном 5 мс, задержке вращения в 4 мс, скорости передачи данных 8 Мбайт/с и среднем размере файла 8 Кбайт, сколько понадобится времени для того, чтобы прочитать файл в оперативную память, а затем записать его обратно на новое место на диске? При тех же параметрах сколько потребуется времени для уплотнения половины 16-гигабайтного диска?
14. В свете ответа на предыдущий вопрос, есть ли вообще смысл в уплотнении диска?
15. Некоторым цифровым потребительским устройствам требуется хранить данные, например, в виде файлов. Назовите современное устройство, для которого хранение данных в виде непрерывных файлов является прекрасной идеей.
16. Как реализован произвольный доступ к файлам в MS-DOS?
17. Рассмотрите i-узел, изображенный на рис. 6.12. Если он содержит 10 дисковых адресов, по 4 байт каждый, а все дисковые блоки имеют размер 1024 байт, чему равен максимальный размер файла?

18. Было предложено увеличить производительность и эффективность использования дискового пространства при помощи хранения коротких файлов прямо в i-узлах. Исходя из рис. 6.12, сколько данных может храниться внутри i-узла?
19. Две студентки с факультета кибернетики, Кэролин и Элинор, обсуждают i-узлы. Кэролин утверждает, что память стала такой большой и дешевой, что при открытии файла проще и быстрее считать новую копию i-узла в таблицу i-узлов, чем искать этот i-узел по всей таблице. Элинор не согласна. Кто прав?
20. Назовите одно преимущество жестких связей над символьными связями и одно преимущество символьных связей над жесткими связями.
21. Учет свободного дискового пространства может осуществляться с помощью связных списков или битовых массивов. Дисковые адреса состоят из D бит. При каком условии для диска из B блоков, F из которых свободны, список займет меньше места, чем битовый массив? Выразите ваш ответ в процентах от объема диска для $D = 16$.
22. После первого форматирования дискового раздела начало битового массива учета свободных блоков выглядит так: 1000 0000 0000 0000 (первый блок используется для корневого каталога). Система всегда ищет свободные блоки от начала раздела, поэтому после записи файла A, занимающего 6 блоков, битовый массив принимает следующий вид: 1111 1110 0000 0000. Покажите, как будет выглядеть битовый массив после каждой из следующих действий:
    а) записывается файл B размером в 5 блоков;
    б) удаляется файл A;
    в) записывается файл C размером в 8 блоков;
    г) удаляется файл B.
23. Что произойдет, если битовый массив или список свободных блоков окажется полностью потерян в результате сбоя? Есть ли способ восстановления от такого сбоя или с диском можно попрощаться? Обсудите свой ответ отдельно для файловых систем UNIX и FAT-16.
24. Ночная работа Оливера Аула в университете состоит в смене лент, используемых для архивации данных. Ожидая окончания записи на каждую ленту, Оливер пишет статью, в которой доказывает, что пьесы Шекспира были написаны инопланетными пришельцами. За неимением другой системы, его текстовый процессор работает прямо в архивируемой системе. Есть ли какие-либо проблемы, связанные с подобной организацией?
25. В главе обсуждалась тема создания инкрементных резервных копий. В системе Windows легко определить, какой файл следует архивировать, так как у каждого файла имеется специальный архивный бит. В системе UNIX такого бита нет. Как программа архивации в системе UNIX определяет, для какого файла следует создать резервную копию?
26. Допустим, что файл 21 на рис. 6.21 не был изменен с момента последней архивации. Как будут при этом отличаться четыре битовых массива на рис. 6.22?

27. Было предложено хранить первую часть каждого файла системы UNIX в том же дисковом блоке, что и его i-узел. Каковы преимущества такого подхода?
28. Рассмотрим рис. 6.23. Возможна ли ситуация, при которой для некоего блока значение счетчиков в *обоих* списках равно 2? Как может быть исправлена эта проблема?
29. Производительность файловой системы зависит от процента блоков, которые удается в нем найти. Напишите формулу для среднего времени удовлетворения запроса блока при частоте успешных обращений, равной h, если удовлетворение запроса из кэша занимает 1 мс, а для считывания блока с диска требуется 40 мс. Нарисуйте график этой зависимости для значений h в интервале от 0 до 1,0.
30. У гибкого диска 40 цилиндров. Операция поиска занимает 6 мс на цилиндр. Если не пытаться разместить блоки файла близко друг к другу, два логически последовательных блока (то есть следующих один за другим в файле) окажутся в среднем на расстоянии 13 цилиндров друг от друга. Однако если операционная система пытается объединять логически соседние блоки в кластеры, то среднее межблоковое расстояние может быть уменьшено (например) до двух цилиндров. Сколько понадобится времени в обоих случаях для считывания 100-блочного файла, если задержка вращения составляет 100 мс, а время переноса одного блока равно 25 мс?
31. Рассмотрите идею графика на рис. 6.17, но для диска со средним временем поиска 8 мс, скоростью вращения 15 000 об/мин и 262 144 байт на дорожке. Чему равна скорость передачи данных для блоков размера 1, 2 и 4 Кбайт?
32. Некая файловая система использует 2-килобайтные дисковые блоки. Медианный размер файлов составляет 1 Кбайт. Если бы все файлы имели размер в 1 Кбайт, какая часть диска терялась бы понапрасну? Как вы думаете, потери в реальной системе выше этого числа или ниже? Поясните свой ответ.
33. Система CP/M разрабатывалась для работы на маленьком гибком диске в качестве основного устройства хранения данных. Предположим, ее следует перенести на современный компьютер с большим жестким диском. Чему равен максимальный размер жесткого диска, при котором не потребуются изменения формата каталога, показанного на рис. 6.28? Поля в каталоге и другие системные параметры могут быть при необходимости изменены. Какие изменения вы бы сделали?
34. Таблица FAT-16 системы MS-DOS содержит 64 К элементов. Предположим, что один бит потребовался для других целей и что таблица вместо 64 К содержит 32 768 элементов. Чему в этих условиях будет равен максимальный размер файла при отсутствии других изменений?
35. В системе MS-DOS файлам приходится соревноваться за место в таблице FAT-16, хранящейся в оперативной памяти. Если один файл использует k элементов, то есть k элементов, недоступных другим файлам, какие ограничения это накладывает на общую длину всех вместе взятых файлов?

36. В файловой системе UNIX килобайтные блоки и 4-байтовые дисковые адреса. Чему равен максимальный размер файла, если i-узел содержит 10 прямых адресов и по одному одинарному, двойному и тройному косвенному элементу?

37. Сколько понадобится дисковых операций для считывания i-узла файла */usr/ast/courses/os/handout.t*? Предположим, что i-узел корневого каталога находится в оперативной памяти, но больше ничего, относящегося к этому пути, в памяти нет. Кроме того, предположите, что все каталоги занимают по одному блоку диска.

38. Во многих версиях системы UNIX i-узлы хранятся в начале диска. Альтернативный дизайн заключается в выделении i-узлу блока в момент создания файла и помещении этого блока в начале первого блока файла. Обсудите преимущества и недостатки обоих подходов.

39. Напишите программу, изменяющую порядок байтов в файле, так что последний байт становится первым, и наоборот. Программа должна работать с файлами произвольной длины, однако постарайтесь сделать программу достаточно эффективной.

40. Напишите программу, начинающую работу в заданном каталоге и спускающуюся по дереву каталогов, записывая по пути размеры всех встретившихся ей файлов. Закончив сканирование каталога, программа должна распечатать гистограмму размеров файлов, используя шаг гистограммы в качестве параметра (например, при шаге 1024, файлы размером от 0 до 1023 байт попадают в один интервал, от 1024 до 2047 байт — в следующий интервал и т. д.).

41. Напишите программу, сканирующую все каталоги файловой системы UNIX и находящую все i-узлы с двумя и более жесткими связями. Для каждого такого файла программа должна печатать список всех имен файлов, указывающих на этот файл.

42. Напишите новую версию программы *ls* для системы UNIX. Эта версия должна принимать в качестве параметра одно или несколько имен каталогов и для каждого каталога выдавать список всех файлов, содержащихся в этом каталоге, по одной строке на файл. Каждое поле должно форматироваться в соответствии с его типом. Укажите в списке только первый дисковый адрес или не указывайте никаких.

43. Напишите файловую систему CP/M на языке C или C++. Необходимую для этого информацию вы можете найти в Интернете.

Глава 7
Мультимедийные операционные системы

В последнее время все более широкое распространение получают такие компьютерные приложения, как цифровые фильмы, видеоклипы и музыка. Аудио- и видеофайлы могут храниться на диске и воспроизводиться по требованию. Характеристики этой цифровой информации сильно отличаются от характеристик традиционных текстовых файлов, для работы с которыми создавались современные файловые системы. Для управления новыми типами данных требуются новые файловые системы. Кроме того, сохранение и воспроизведение аудио и видео накладывает новые требования на планировщик и другие части операционной системы. В следующих разделах мы изучим многие из этих вопросов и их влияние на устройство операционных систем, созданных для управления мультимедиа.

Как правило, цифровые фильмы называют **мультимедиа**, что буквально означает «более чем один носитель информации». Если следовать этому определению, то данная книга также является мультимедийным трудом, так как содержит информацию двух различных типов: текст и изображения (рисунки). Но большинство людей обычно употребляют это слово, имея в виду документ, содержащий средства информации, протяженные во времени, то есть проигрываемые в течение определенного интервала времени. В данной книге мы будем использовать это слово именно в таком значении.

Другой неоднозначный термин — это «видео». Технически это просто графическая составляющая фильма (в противоположность звуковой составляющей). В самом деле, у видеокамер и телевизоров часто есть два разъема, один из которых помечен словом «видео», а другой — «аудио», поскольку эти два сигнала разделены. Однако термин «цифровое видео» обычно относится к полному продукту, содержащему как изображение, так и звук. Ниже для обозначения полного продукта будет использоваться термин «фильм». Обратите внимание, что термин «фильм», употребляемый в этом смысле, не обязан быть двухчасовой лентой, снятой на студии в Голливуде за цену, превышающую стоимость Боинга 747. 30-секундный клип новостей, загружаемый по Интернету с домашней web-страницы CNN, также является фильмом в нашем определении. Говоря об очень коротких фильмах, мы будем употреблять и термин «видеоклип».

Введение в мультимедиа

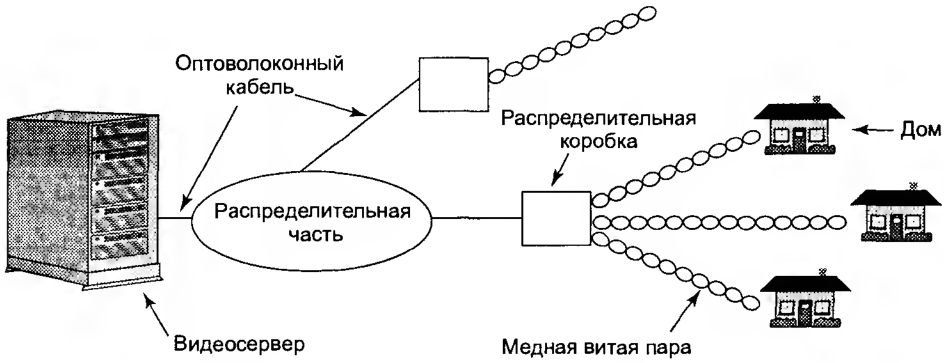
Прежде чем перейти к обсуждению технологии мультимедиа, следует сказать несколько слов о его сегодняшнем состоянии и перспективах с точки зрения пользователя. На одиночном компьютере мультимедиа часто означает воспроизведение ранее записанного фильма с **DVD** (Digital Versatile Disk — универсальный цифровой диск). DVD представляют собой оптические диски, для производства которых используются те же самые поликарбонатные (пластиковые) диски диаметром 120 мм, что и для производства CD-ROM, но информация записывается на них с большей плотностью. Емкость DVD составляет от 5 до 17 Гбайт, в зависимости от формата.

Другой вариант использования мультимедиа заключается в загрузке видеоклипа из Интернета. Многие web-страницы содержат ссылки для загрузки коротких фильмов. На скорости 56 кбит/с загрузка даже короткого видеоклипа занимает довольно долгое время, однако с появлением более совершенных технологий передачи данных, таких как кабельное телевидение и **ADSL** (Asymmetric Digital Subscriber Line — асимметричная цифровая абонентская линия), видеоклипов в Интернете становится все больше.

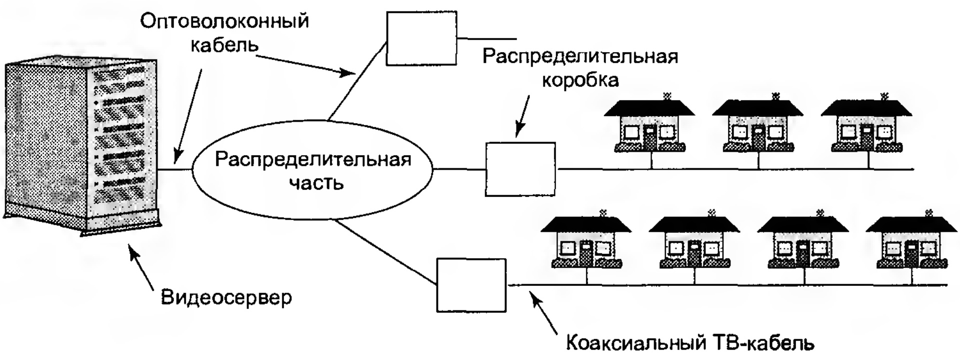
Еще одна область, в которой мультимедийные средства должны поддерживаться, — это создание самих видеофильмов. Существуют различные системы редактирования мультимедиа, для более высокой производительности которых требуются специальные операционные системы, поддерживающие мультимедиа помимо традиционных задач.

Все более важное положение мультимедиа занимает в компьютерных играх. В играх часто проигрываются небольшие видеоклипы, иллюстрирующие некоторые события. Эти клипы, как правило, короткие, но в игре их содержится большое количество, и нужный клип выбирается динамически, в зависимости от действия пользователя. Сложность таких клипов возрастает с каждым годом.

Наконец, наиболее интересной областью применения мультимедиа является **видео по заказу**, под которым подразумевается возможность для абонента не выходя из дома выбрать фильм для просмотра на своем телевизоре и тут же начать его просмотр. Для реализации видео по заказу требуется специальная инфраструктура. На рис. 7.1 показаны два возможных варианта инфраструктуры видео по заказу. Каждая инфраструктура состоит из одного или нескольких видеосерверов, распределительной сети и телевизионных приставок в каждом доме для декодирования сигнала. **Видеосервер** представляет собой мощный компьютер, хранящий в своей файловой системе большое количество фильмов и воспроизводящий их по требованию. Иногда в качестве видеосерверов используются мэйнфреймы, так как подключить, скажем, 1000 больших дисков к мэйнфрейму не составляет сложности, тогда как подключение 1000 дисков к персональному компьютеру любого типа представляет собой серьезную проблему. Большая часть материала следующих разделов посвящена видеосерверам и их операционным системам.



а



б

Рис. 7.1. Видео по заказу с применением различных технологий местного распределения: ADSL (*а*); кабельное телевидение (*б*)

Распределительная сеть между пользователями и видеосервером должна быть способна передавать данные на высокой скорости в режиме реального времени. Устройство таких сетей интересно и сложно, но оно не вмещается в рамки этой книги. Мы более не будем упоминать о них, только отметим, что в этих сетях всегда используются оптоволоконные кабели от видеосервера до того места, где живут абоненты. В системах ADSL, предоставляемых телефонными компаниями, последний километр данные передаются по существующим витым парам телефонных линий. В системах кабельного телевидения, услуги которого предоставляются операторами кабельной связи, для локального распределения используются существующие телевизионные кабели. Преимущество системы ADSL заключается в том, что каждому пользователю предоставляется выделенный канал с гарантированной пропускной способностью. Недостатком является низкая пропускная способность (несколько мегабит в секунду), что вызвано ограничениями существу-

ющих телефонных линий. Кабельное телевидение использует высокоскоростные коаксиальные кабели (гигабиты в секунду), однако нескольким пользователям приходится совместно использовать один кабель, что приводит к состязанию за кабель и не гарантирует пропускной способности отдельному пользователю.

Последний узел системы представляет собой **телевизионную приставку**, к которой и присоединен ADSL или телевизионный кабель. Это устройство является в действительности нормальным компьютером со специальными микросхемами для декодирования и декомпрессии видеопотока. Как минимум он содержит центральный процессор, оперативную память, ПЗУ и интерфейс с системой ADSL или телевизионным кабелем.

Вместо телевизионной приставки фильм можно просматривать и на мониторе имеющегося у клиента персонального компьютера. В чем же причина внимания к телевизионным приставкам, хотя у большинства абонентов уже есть персональные компьютеры? Операторы видео по заказу ожидают, что их клиенты захотят смотреть фильмы в гостиных, в которых обычно есть телевизор, но нет компьютера. С технической точки зрения использование персонального компьютера вместо телевизионной приставки имеет гораздо больше смысла, так как компьютер обладает большими возможностями, у него есть диск большого объема и дисплей с гораздо более высоким разрешением. Мы часто будем использовать различие между видеосервером и клиентским процессом на стороне пользователя, занимающимся декодированием и отображением фильма. Однако с точки зрения конструкции системы почти не имеет значения, работает ли клиентский процесс на персональном компьютере или в телевизионной приставке. В настольной системе редактирования видеоизображения все процессы работают на одной и той же машине, но мы будем продолжать использовать терминологию сервера и клиента, чтобы было ясно, что делает каждый процесс.

Возвращаясь к мультимедиа, стоит отметить две ключевые характеристики, понимание которых необходимо для успешной работы:

1. Мультимедиа использует предельно высокие скорости передачи данных.
2. Для мультимедиа требуется воспроизведение в режиме реального времени.

Высокие скорости передачи данных обусловлены природой визуальной и акустической информации. Человеческий глаз и ухо способны обрабатывать за секунду огромные объемы данных, поэтому им необходимо поставлять информацию с той скоростью, которая обеспечит приемлемый уровень качества восприятия. Скорости передачи данных некоторых цифровых источников мультимедиа и некоторых распространенных устройств приведены в табл. 7.1. (Обратите внимание, что 1 Мбит/с — это 10^6 бит/с, но 1 Гбайт — это 2^{30} байт.) Некоторые из этих форматов кодирования данных будут обсуждаться ниже в этой главе. Следует обратить внимание на высокую скорость передачи данных, требующуюся для мультимедиа, необходимость сжатия данных и на большие объемы, занимаемые данными. Например, несжатый 2-часовой фильм в формате HDTV (High Definition TeleVision — телевидение высокой четкости) занимает 570 Гбайт. Видеосерверу, хранящему 1000 таких фильмов, нужно 570 Тбайт дискового пространства, что совсем нетривиально с точки зрения современных стандартов. Также следует отметить, что при таких скоростях современная аппаратура не способна обходиться без сжатия данных. Сжатие данных будет рассматриваться ниже в этой главе.

Таблица 7.1. Скорости передачи данных некоторых мультимедийных устройств и некоторых высокоскоростных устройств ввода-вывода

Источник	Мбит/с	Гбайт/ч	Устройство	Мбит/с
Телефон (кодово-импульсная модуляция)	0,064	0,03	Быстрая сеть Ethernet	100
MP3 музыка	0,14	0,06	EIDE диск	133
Аудиокомпакт-диск	1,4	0,62	Сеть ATM OC-3	156
MPEG-2 фильм (640×480)	4	1,76	SCSI UltraWide диск	320
Цифровая видеокамера (720×480)	25	11	IEEE 1394 (FireWire)	400
Несжатое телевидение (640×480)	221	97	Гигабитная сеть Ethernet	1000
Несжатое HDTV (1280×720)	648	288	SCSI Ultra-160 диск	1280

Второе требование, накладываемое мультимедийными приложениями на систему, заключается в необходимости доставки данных в режиме реального времени. Графическая составляющая видеофильма состоит из последовательности кадров, передаваемых с определенной частотой. Система NTSC, используемая в Северной и Южной Америке и Японии, работает с частотой 30 кадров в секунду (точнее, 29,97), тогда как в системах PAL и SECAM, используемых в остальном мире, применяется частота 25 кадров в секунду (25,00 для любителей точности). Кадры должны доставляться через точные интервалы времени по 33,3 мс или по 40 мс соответственно, чтобы изображение не подергивалось.

Официально сокращение NTSC означает National Television Standards Committee (Национальный комитет по телевизионным стандартам США), однако плохое качество передачи цвета на ранних этапах существования телевидения привело к появлению шутки, согласно которой NTSC следует расшифровывать как «постоянно меняющийся цвет» (Never Twice the Same Color). Аббревиатура PAL расшифровывается как Phase Alternation Line (построчное изменение фазы). Технически это лучшая из систем. Сокращение SECAM означает SEquentiel Couleur Avec Memoire (последовательные цвета с запоминанием). Эта система была разработана во Франции для защиты французских производителей телевизоров от иностранных конкурентов. Система SECAM также используется в восточной Европе; при появлении там телевидения тогдашние коммунистические правительства не хотели допустить, чтобы их граждане смотрели немецкое (PAL) телевидение, поэтому они выбрали несовместимую систему.

Чувствительность человеческого уха превосходит чувствительность глаза, поэтому отклонение во времени доставки даже в несколько миллисекунд будет заметным. Неравномерность времени доставки называется **джиттером**. Для обеспечения высокого качества воспроизведения следует удерживать джиттер в строгих рамках. Обратите внимание, что джиттер — это не то же самое, что задержка. Если распределительная сеть на рис. 7.1 задерживает при передаче все биты ровно на 5000 с, фильм начнется несколько позже, но будет выглядеть прекрасно. С другой стороны, если задержка кадров будет случайной величиной в пределах от 100 до 200 мс, видео будет выглядеть как старый фильм Чарли Чаплина, независимо от того, кто играет главные роли.

Параметры реального времени, требуемые для приемлемого воспроизведения мультимедиа, часто называют параметрами **качества обслуживания**. К ним отно-

сят среднюю доступную пропускную способность, максимальную пропускную способность, минимальную и максимальную задержку (что вместе ограничивает джиттер) и вероятность потери бита. Например, сетевой оператор может предлагать службу, гарантирующую среднюю пропускную способность, равную 4 Мбит/с, 99 % задержек при передаче в интервале от 105 до 110 мс и частоту потерь битов, равную 10^{-10}, что будет являться прекрасными параметрами для передачи фильма в формате MPEG-2. Оператор может также предоставлять более дешевую, худшего качества службу, со средней пропускной способностью в 1 Мбит/с (например, ADSL). В этом случае качество фильма придется каким-то образом снизить, скажем, снизив разрешение или частоту кадров, либо отбросив информацию о цвете и передавая фильм в черно-белом изображении.

Наиболее простой способ обеспечения гарантированного качества службы заключается в предварительном резервировании мощностей для каждого нового клиента. Резервируемые ресурсы включают в себя часть времени центрального процессора, буферы памяти, пропускную способность диска и пропускную способность сети. Если появляется новый клиент, желающий посмотреть фильм, но видеосервер (или сеть) вычисляет, что ему не хватит мощности для еще одного клиента, в этом случае видеосерверу придется отказать новому клиенту, чтобы не снижать качество обслуживания уже обслуживаемых клиентов. Таким образом, мультимедийным серверам требуется схема резервирования ресурсов и **алгоритм управления допуском**, принимающий решение о том, может ли сервер выполнить дополнительную работу.

Мультимедийные файлы

В большинстве систем обычный текстовый файл состоит из линейной последовательности байтов без какой-либо структуры, о которой знала бы операционная система. В мультимедиа ситуация гораздо более сложная. Во-первых, видео- и аудиоданные полностью различны. Они вводятся совершенно разными устройствами (ПЗС-матрицей или микрофоном), у них различная внутренняя структура (видео передается с частотой 25–30 кадров в секунду, тогда как аудио обычно хранится в виде 44 100 отсчетов в секунду), и воспроизводятся они также различными устройствами (монитором и громкоговорителями).

Более того, большинство голливудских фильмов сегодня нацелено на всемирную аудиторию, немалая часть которой не говорит по-английски. Последняя проблема решается одним из двух способов. Для некоторых стран производится дополнительная звуковая дорожка, с голосами (но не звуковыми эффектами), дублированными на местном языке. В Японии все телевизоры оснащены двумя звуковыми каналами, чтобы позволить зрителю слушать зарубежные фильмы на языке оригинала или на японском. Для выбора языка пульты дистанционного управления оснащаются специальной кнопкой. В других странах используется оригинальный звук с субтитрами на местном языке.

Кроме того, многие телевизионные передачи сегодня снабжаются скрытыми субтитрами на английском, что позволяет англо-говорящим, но плохо слышащим людям смотреть эти передачи. В результате цифровой фильм может оказаться

состоящим из большого количества файлов: видеофайла, нескольких аудиофайлов и нескольких текстовых файлов с субтитрами на различных языках. DVD способны хранить до 32 звуковых дорожек на разных языках, а также файлы с субтитрами. Простой набор мультимедийных файлов проиллюстрирован на рис. 7.2. Значение файлов быстрой перемотки вперед и назад будет объяснено ниже в этой главе.

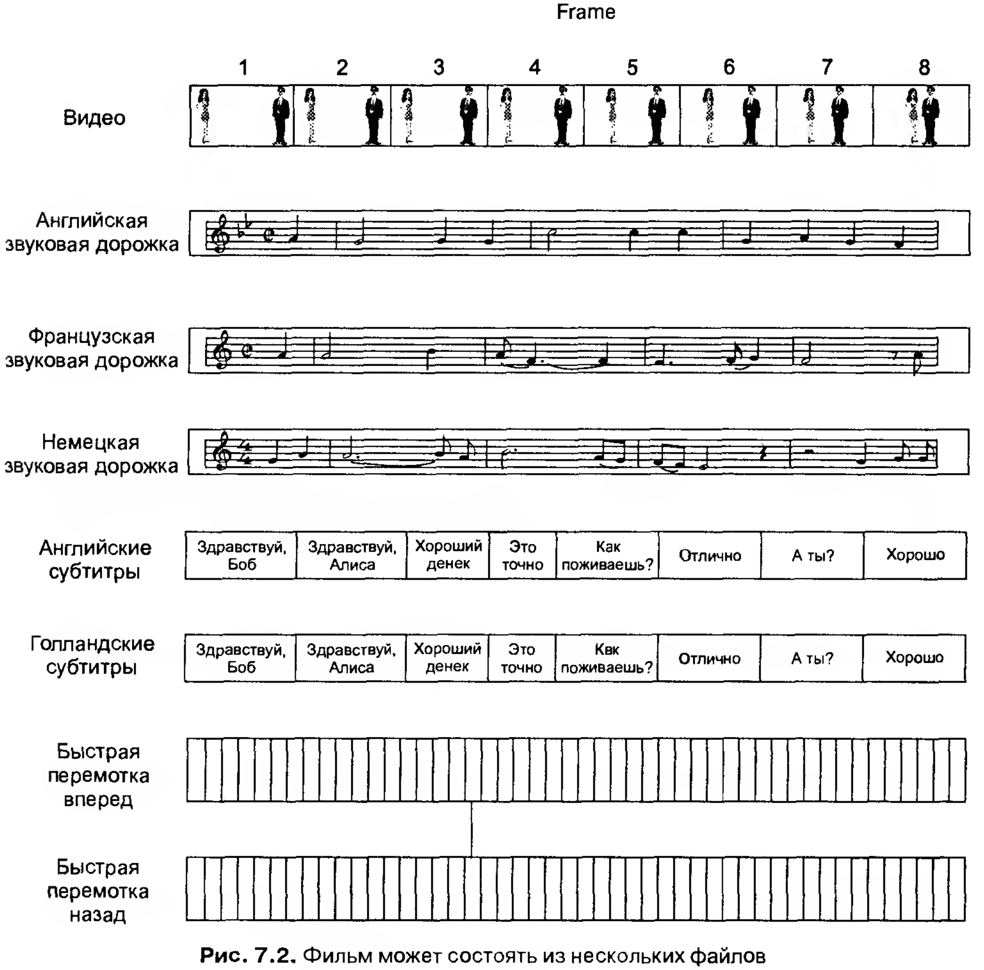


Рис. 7.2. Фильм может состоять из нескольких файлов

Таким образом, файловая система должна следить за несколькими «субфайлами». Одна возможная схема заключается в том, что каждый субфайл хранится как обычный файл (например, при помощи i-узла, в котором хранятся номера всех его блоков), а все субфайлы описываются новой структурой. Другой способ состоит в создании новой разновидности двумерного i-узла, в котором в каждой колонке перечисляются блоки каждого субфайла. В общем, организация должна предоставлять зрителю возможность при просмотре фильма динамически выбирать звуковые дорожки и субтитры.

Во всех случаях необходим некий способ синхронизации субфайлов, чтобы любая звуковая дорожка соответствовала изображению. Если изображение и звук хотя бы слегка рассинхронизируются, зритель будет слышать слова актера до или после движения его губ, что легко определяется и довольно сильно раздражает зрителя.

Для лучшего понимания организации мультимедийных файлов необходимо понимать, как работают цифровое аудио и видео. В следующем разделе мы познакомимся с основами цифрового представления аудио- и видеосигналов.

Кодирование звука

Аудиоволна (звуковая волна) представляет собой одномерную акустическую волну. Когда акустическая волна попадает в ухо человека, барабанная перепонка начинает вибрировать, вызывая вибрацию тонких костей среднего уха, в результате чего в мозг по нерву посылается пульсирующий сигнал. Эта пульсация воспринимается слушателем как звук. Подобным образом, когда акустическая волна воздействует на микрофон, микрофон формирует электрический сигнал, представляющий амплитуду звука в виде функции времени.

Человеческое ухо слышит сигналы в диапазоне частот от 20 до 20 000 Гц, хотя некоторые животные, например собаки, могут слышать и более высокие частоты. Наше ухо слышит логарифмически, поэтому сила звука обычно измеряется в логарифмах отношения амплитуд, например в **децибелах (дБ)**:

$$\text{дБ} = 20 \log_{10}(A/B).$$

Если принять нижний предел слышимости (давление около 0,0003 дин/см^2, что равно 3×10^{-5} Па) для синусоидальной волны частотой в 1 кГц за 0 дБ, то громкости обычного разговора будет соответствовать 50 дБ, а болевой порог наступит при силе звука около 120 дБ, что соответствует отношению амплитуд, равному одному миллиону. Чтобы избежать путаницы, A и B в формуле являются *амплитудами*. Если использовать вместо амплитуд уровни мощности, которые пропорциональны квадратам амплитуд, следует в приведенной выше формуле вместо коэффициента 20 поставить число 10.

Аудиоволны могут преобразовываться в цифровую форму при помощи **аналогово-цифрового преобразователя (АЦП)**. АЦП принимает на входе электрическое напряжение и формирует двоичное число на выходе. На рис. 7.3, *а* показан пример синусоидальной волны. Чтобы представить этот сигнал в цифровом виде, мы можем измерять значения сигнала (отсчеты) через равные интервалы времени, как показано на рис. 7.3, *б*. Если звуковая волна не является чисто синусоидальной, а представляет собой сумму нескольких синусоидальных волн, самая высокая частота составляющих которых равна f, тогда для последующего восстановления сигнала достаточно измерять значения сигнала с частотой дискретизации $2f$. Это утверждение было математически доказано Найквистом в 1924 году. Производить измерения сигнала с большей частотой нет смысла, так как более высокие частоты отсутствуют в сигнале.

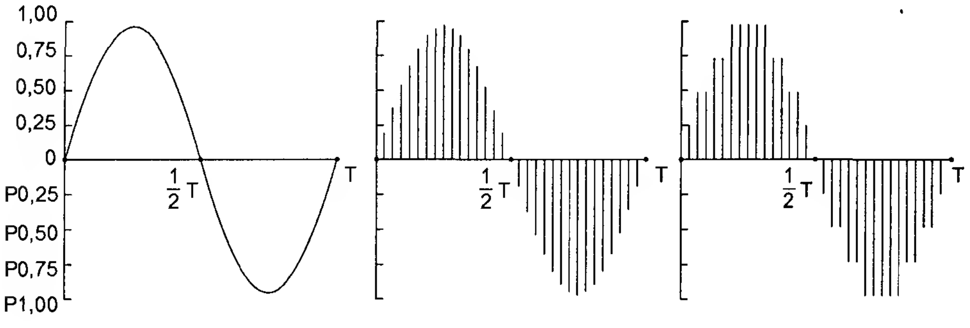


Рис. 7.3. Синусоидальная волна (*а*); дискретизация синусоидальной волны (*б*); квантование отсчетов (*а*)

Оцифрованные отсчеты никогда не бывают точными. Трехбитовые отсчеты на рис. 7.3 могут принимать только 8 значений, от –1,00 до +1,00 с шагом 0,25. При 8-битовом квантовании каждый отсчет может принимать одно из 256 различных значений. При 16 битах на отсчет можно кодировать сигнал с еще более высокой точностью, так как каждому значению сигнала можно поставить в соответствие одно из 65 536 различных значений. Ошибка, возникающая в результате неточного соответствия квантованного сигнала исходному сигналу, называют **шумом квантования**. При недостаточном количестве битов, которыми представляется каждый отсчет сигнала, этот шум может быть настолько велик, что будет различим на слух как искажение исходного сигнала или как посторонние шумы.

Двумя хорошо известными примерами оцифрованного звука являются телефон (новые цифровые АТС) и аудиокомпакт-диски. В **кодово-импульсной модуляции**, применяемой для телефонной системы, используются 7-битовые (в Северной Америке и Японии) и 8-битовые (в Европе) отсчеты, передаваемые 8000 раз в секунду. Таким образом, получаемая скорость передачи данных составляет 56 000 бит/с или 64 000 бит/с. При частоте дискретизации в 8 кГц частотные составляющие сигнала выше 4 кГц теряются.

Аудиокомпакт-диски содержат звуковой сигнал, оцифрованный с частотой дискретизации 44 100 Гц, в результате чего они могут хранить звуки с частотами до 22 кГц, что достаточно для людей, но плохо для собак. Каждому отсчету выделяется 16 бит, которые используются как обычное 2-байтовое целое число, пропорциональное амплитуде сигнала. Обратите внимание, что 16-битовый отсчет может принимать всего 65 536 различных значений (96 дБ), хотя динамический диапазон уха составляет около одного миллиона (120 дБ). Тем не менее этот формат достаточно хорош, а во многих случаях даже избыточен. При 44 100 отсчетов в секунду по 16 бит каждый аудиокомпакт-диску нужны пропускная способность в 705,6 кбит/с для монофонического сигнала и 1,411 Мбит/с для стереофонического. Алгоритм MPEG, уровень 3 (MP3) позволяет сжимать оцифрованный звук примерно в 10 раз. В последние годы стали популярными переносные плейеры, воспроизводящие музыку в этом формате.

Цифровой звук легко обрабатывать на компьютере. Существуют десятки программ для персональных компьютеров, позволяющие пользователям записывать,

воспроизводить, редактировать, микшировать и хранить звук. Сегодня вся профессиональная звукозапись и редактирование звука осуществляется в цифровом виде.

Кодирование изображения

Сетчатка глаза человека обладает инерционными свойствами, то есть яркое изображение, быстро появившееся на сетчатке, остается на ней несколько миллисекунд, прежде чем угаснуть. Если последовательность одинаковых или близких изображений появляются и исчезают с достаточно высокой частотой, то глаз человека не заметит, что он смотрит на дискретные изображения. Частота, при которой глаз перестает замечать мигание яркости источника света (изображения), составляет около 50 Гц. Все видео (то есть телевизионные) системы используют этот принцип для создания двигающихся изображений.

Чтобы понять, как работают видеосистемы, лучше всего начать с простого старомодного черно-белого телевидения. Для преобразования двумерного изображения в вид одномерной зависимости напряжения от времени камера быстро сканирует электронным лучом изображение, разбивая его на горизонтальные линии и записывая по мере продвижения интенсивность света. Закончив сканирование **кадра**, луч возвращается в исходную точку. Путь сканирования, который проходит электронный луч в передающей камере и принимающей телевизионной трубке, показан на рис. 7.4. В последнее время все большее распространение получают матричные жидкокристаллические мониторы и цифровые камеры с ПЗС-матрицами (прибор с зарядовой связью). В этих устройствах нет электронно-лучевой трубки и сканирующего луча.

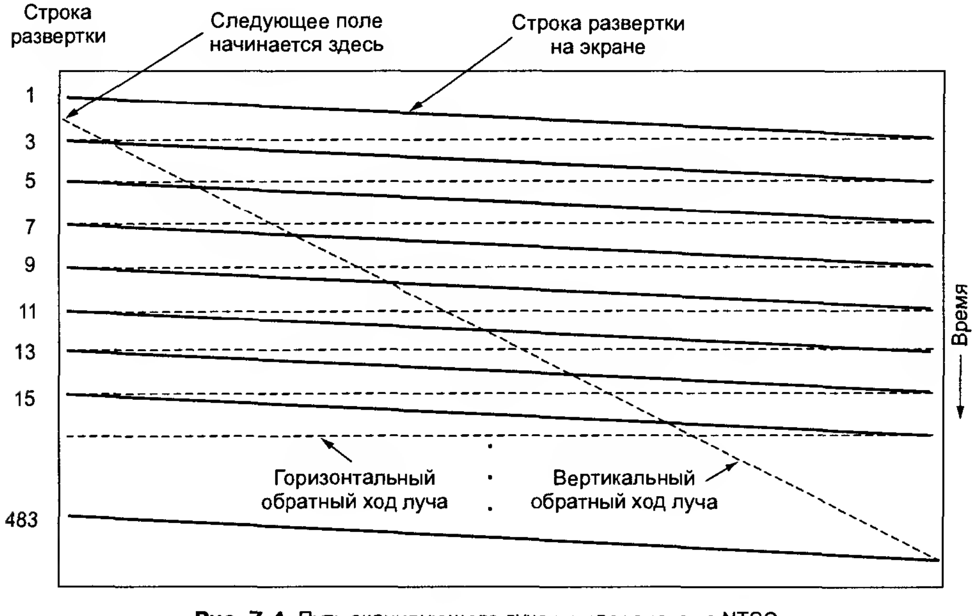


Рис. 7.4. Путь сканирующего луча в видеосистеме NTSC

В разных странах приняты различные стандарты, описывающие параметры сканирования (количество строк развертки и т. д.). В системе NTSC, принятой в Северной и Южной Америке, а также в Японии, экран разбивается на 525 горизонтальных линий развертки, соотношение горизонтального и вертикального размеров экрана составляет 4:3, кадры передаются с частотой 30 кадров в секунду. Принятая в Европе система PAL/SECAM разбивает кадр на 625 линий, размеры экрана у нее также 4:3, а частота кадров составляет 25 кадров в секунду. В обеих системах самые верхние и самые нижние линии кадра не показываются (это связано с круглой формой электронно-лучевой трубки). На экране телевизоров показываются только 483 из 525 линий развертки для системы NTSC и 576 из 625 для системы PAL/SECAM.

Хотя частоты в 25 кадров в секунду достаточно, чтобы передать плавное движение, при такой частоте кадровой развертки многие зрители (особенно пожилые) заметят мигание изображения, связанное с тем, что сетчатка глаза успеет восстановиться, прежде чем появится новый кадр. Увеличение частоты кадров потребовало бы увеличения объемов хранимой и передаваемой информации. Вместо этого было выбрано другое решение. Линии развертки показываются на экране телевизора не подряд, а через одну: сначала все нечетные, а затем все четные. Каждый такой полукадр называют **полем**. Эксперименты показали, что хотя люди замечают мерцание при 25 кадров в секунду, при 50 кадров в секунду оно уже не заметно. Такая техника называется **чересстрочной разверткой**. Телевидение или видеосистема, в которой не используется чересстрочная развертка, называется **поступательным**.

В цветном видео используется тот же принцип развертки, что и в черно-белом, с той разницей, что вместо одного луча изображение представляется синхронно двигающимися тремя лучами: красным, зеленым и синим (RGB — red, green, blue). Комбинации этих трех цветов оказывается достаточно для передачи любого цвета благодаря особенностям устройства человеческого глаза. При передаче по каналу связи эти три цветовых сигнала объединяются в единый **смешанный** сигнал.

Для совместимости с черно-белыми телевизионными приемниками во всех трех системах сигналы RGB линейно объединяются в сумму этих сигналов, называемую **яркостью**, и два сигнала **цветности**. Однако в каждой из систем эти сигналы формируются с использованием различных коэффициентов. Экспериментально доказано, что человеческий глаз обладает гораздо более высокой чувствительностью к изменению сигнала яркости, чем к изменению сигнала цветности. В результате сигнал цветности не обязательно кодировать так же точно, как и сигнал яркости. Поэтому было решено передавать сигнал яркости в том же формате, что и черно-белый сигнал, а два узкополосных сигнала цветности передаются отдельно на более высокой частоте. Телевизоры обычно снабжаются регуляторами яркости и насыщенности цвета. Знакомство с сигналами яркости и цветности важно для понимания принципов действия алгоритмов видеосжатия.

До сих пор мы рассматривали аналоговое видео. Обсудим теперь цифровое видео. Простейшая форма представления цифрового видео заключается в последовательности кадров, состоящих из прямоугольной сетки элементов рисунка, называемых **пикселами**. В цветном телевидении достаточно использовать по 8 бит на каждый из трех цветов RGB, что дает в сумме 24 бита на пиксел. Хотя числом, состоящим из 24 бит, можно обозначить около 16 млн цветовых оттенков, челове-

ческий глаз не в состоянии даже различить такое огромное количество цветовых оттенков, не говоря уже о больших количествах.

Для гладкой передачи движения, как и в аналоговом видео, в цифровом видео необходимо отображать по меньшей мере 25 кадров в секунду. Однако, поскольку качественные мониторы обычно сами сканируют по нескольку раз хранящиеся в их памяти изображения с частотой 75 и более кадров в секунду, проблема мерцания изображения решается на уровне монитора сама собой, и чередование строк в цифровом видео не требуется.

Другими словами, плавность движущегося изображения определяется количеством *отличающихся* изображений в секунду, тогда как мерцание зависит от частоты перерисовки экрана. Не следует путать эти два параметра. Неподвижное изображение, рисуемое с частотой 20 кадров в секунду, не будет дергаться, но будет мерцать, поскольку возбуждение сетчатки глаза успеет угаснуть, прежде чем появится следующий кадр. Фильм, в котором выводится 20 различных кадров в секунду, каждый из которых повторяется по четыре раза, не будет мерцать, но будет заметно отсутствие плавности движений.

Важность этих двух параметров становится ясна, если мы попробуем оценить пропускную способность, необходимую для передачи цифрового видеосигнала по сети. Во всех современных компьютерных мониторах используется соотношение размеров экрана 4:3, поэтому они вполне могут использоваться для отображения телевизионного изображения. Стандартные используемые разрешения экрана составляют 640 × 480 (VGA), 800 × 600 (SVGA) и 1024 × 768 (XGA). Для показа цифрового видео на XGA-экране при 24 битах на пиксел и 25 кадрах в секунду потребуется поток данных со скоростью 472 Мбит/с. Даже канала OC-9 будет недостаточно для этого, а прокладка такого кабеля в каждый дом пока еще не стоит на повестке дня. Удвоение частоты, чтобы избавиться от мерцания, выглядит еще менее привлекательно. Лучшее решение состоит в том, чтобы передавать 25 кадров в секунду и позволить компьютеру самому хранить эти кадры и отображать их с утроенной или учетверенной частотой. В обычном широковещательном телевидении такая стратегия почти не используется[1], так как у обычных телевизоров нет памяти, кроме того, для хранения аналоговых сигналов в ОЗУ необходимо сначала преобразовать их в цифровую форму, что потребует дополнительного оборудования. Таким образом, чередование строк применяется в обычном широковещательном телевидении, но оно не нужно в цифровом видео.

Сжатие видеоинформации

Итак, теперь должно быть очевидно, что о передаче мультимедийной информации в несжатом виде не может быть и речи. К счастью, за несколько последних десятков лет было разработано множество методов сжатия, делающих возможной передачу мультимедийной информации. В данном разделе мы рассмотрим некоторые методы сжатия мультимедийных данных, особенно изображений. Более подробно эта тема освещена в [118, 313].

[1] Кроме дорогих 100-герцевых телевизоров. — *Примеч. перев.*

Для всех систем сжатия требуется два алгоритма: один для компрессии данных у источника информации, а другой — декомпрессии у ее получателя. В литературе эти алгоритмы называют соответственно алгоритмами **кодирования** и **декодирования**. Мы также будем пользоваться здесь этой терминологией.

Эти алгоритмы обладают определенной асимметрией, о которой следует сказать несколько слов. Во-первых, во многих приложениях мультимедийный документ, например фильм, кодируется всего один раз при его создании и помещении на мультимедийный сервер, но декодируется тысячи раз при просмотре пользователями. Эта асимметрия означает, что алгоритм кодирования может быть довольно медленным и требовать дорогого оборудования, тогда как алгоритм декодирования должен быть быстрым и должен работать даже на дешевом оборудовании. С другой стороны, для мультимедиа реального времени, например видеоконференции, медленное кодирование неприемлемо. Кодирование здесь должно происходить на ходу, в режиме реального времени.

Вторая причина асимметрия состоит в том, что процесс кодирования/декодирования не обязан быть обратимым. Это значит, что при сжатии обычного файла, его передаче и декомпрессии получатель обязан получить точную копию оригинала. При передаче мультимедиа абсолютная точность не требуется. Вполне допустимы небольшие отклонения видеосигнала от оригинала после его кодирования и декодирования. Система, в которой декодированный сигнал не точно соответствует кодированному оригиналу, называется системой **с потерями**. Все системы сжатия данных, применяемые в мультимедиа, являются системами с потерями, так как они позволяют достичь гораздо большего коэффициента сжатия.

Стандарт JPEG

Стандарт **JPEG** для сжатия неподвижных изображений с непрерывно меняющимся цветом (например, фотографий) был разработан группой экспертов в области фотографии JPEG (Joint Photography Experts Group — Объединенная группа экспертов по машинной обработке фотоизображений). Эта группа работала под совместным покровительством международного союза телекоммуникаций ITU, Международной организации по стандартизации ISO и еще одной организации, занимающейся стандартизацией, IEC (International Electrotechnical Commission — международная электротехническая комиссия). Стандарт JPEG является очень важным для мультимедиа, так как в первом приближении мультимедийный стандарт для движущихся изображений MPEG представляет собой просто кодирование каждого кадра отдельно алгоритмом JPEG плюс некоторые дополнительные процедуры межкадрового сжатия и обнаружения движения. Стандарт JPEG определен как Международный стандарт 10918. У метода сжатия JPEG четыре режима и множество параметров, но нас здесь будет интересовать только использование стандарта JPEG для кодирования 24-битового RGB-видеоизображения, поэтому мы опустим некоторые незначительные детали.

Первый этап кодирования изображения алгоритмом JPEG представляет собой подготовку блока. Рассмотрим частный случай кодирования 24-битового RGB-видеоизображения размером 640×480 пикселов, как показано на рис. 7.5, *а*. Поскольку разделение на яркость и цветность позволяют сильнее сжать изображение, снача-

ла из значений RGB вычисляются яркость и два значения цветности. В стандарте NTSC они называются *Y*, *I* и *Q* соответственно. В стандарте PAL значения цветности называются *U* и *V* и коэффициенты используются другие, но идея та же самая. Ниже мы будем использовать имена стандарта NTSC.

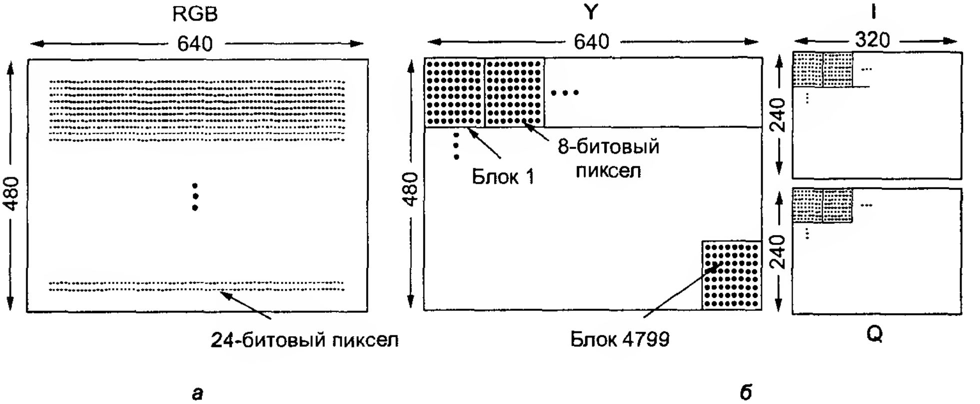


Рис. 7.5. Исходные данные в формате RGB (*а*); после подготовки блока (*б*)

Для значений *Y*, *I* и *Q* строятся отдельные матрицы с элементами в диапазоне от 0 до 255. Затем значения цветности *I* и *Q* усредняются по квадратным блокам из четырех пикселов, что уменьшает размеры матриц цветности в 4 раза до размера 320×240. Это сжатие является преобразованием с потерями, но человеческий глаз его практически не замечает, так как чувствительность глаза к яркости значительно выше, чем к цветности. Но уже на этом этапе общий объем данных уменьшается вдвое. Затем из каждого элемента вычитается число 128, чтобы переместить число 0 в середину диапазона. Наконец, каждая матрица разбивается на квадраты по 8×8 пикселов. Таким образом, матрица *Y* состоит из 4800 квадратных блоков, а матрицы *I* и *Q* содержат по 1200 блоков каждая, как показано на рис. 7.5, *б*.

На втором этапе кодирования изображения алгоритмом JPEG к каждому из 7200 квадратных блоков отдельно применяется дискретное косинусное преобразование. На выходе получается 7200 матриц 8×8 коэффициентов дискретного косинусного преобразования (ДКП). Элемент (0, 0) такой матрицы представляет собой среднее значение блока. Остальные элементы содержат информацию о спектральной мощности каждой пространственной частоты. В теории дискретное косинусное преобразование является преобразованием без потерь (обратимым), но на практике использование чисел с плавающей точкой и трансцендентных функций приводят к ошибкам округления, в результате чего часть информации теряется. Обычно элементы матрицы ДКП-коэффициентов быстро убывают с расстоянием от элемента (0, 0), как показано на рис. 7.6.

По завершении дискретного косинусного преобразования алгоритм JPEG переходит к этапу 3, называемому **квантованием**, в котором наименее важные ДКП-коэффициенты удаляются. Это преобразование (с потерями) выполняется делением всех коэффициентов ДКП-матрицы на табличные весовые коэффициенты. Если все весовые коэффициенты равны 1, то это преобразование ничего не меня-

ет. Однако при резком росте весовых коэффициентов по мере удаления от элемента матрицы (0, 0) более высокие пространственные частоты быстро теряются.

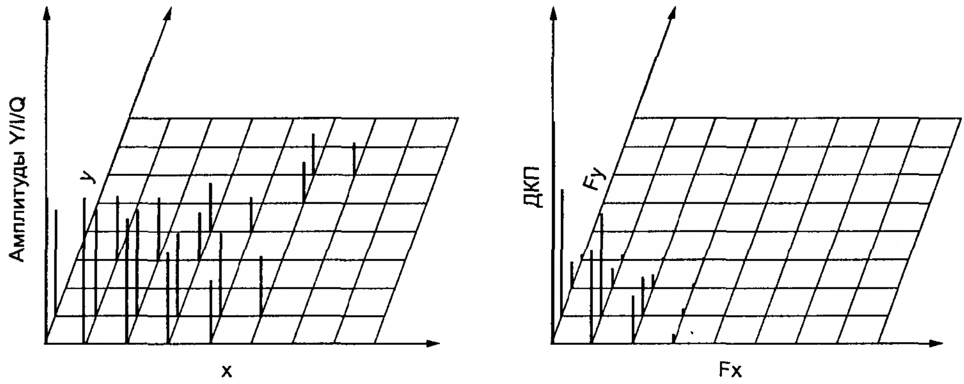


Рис. 7.6. Один блок матрицы Y (*а*); ДКП-коэффициенты (*б*)

Пример этого этапа работы алгоритма показан на рис. 7.7. Здесь мы видим исходную ДКП-матрицу, таблицу квантования и результат деления каждого ДКП-элемента на соответствующий элемент таблицы квантования. Значения в таблице квантования не являются частью стандарта JPEG. Каждое приложение должно содержать свою таблицу весовых коэффициентов, что позволяет ему контролировать соотношение потерь и коэффициента сжатия.

ДКП-коэффициенты

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	2	1	1	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Квантовые коэффициенты

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Таблица квантования

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Рис. 7.7. Квантование ДКП-коэффициентов

На четвертом этапе значение, содержащееся в элементе (0, 0) каждого блока (в левом верхнем углу квадрата), заменяется его отклонением относительно значения в предыдущем блоке. Так как эти значения представляют собой усредненные величины своих блоков, они должны меняться медленно, следовательно, полученные в результате разности должны быть невелики. Для остальных значений разности не вычисляются. Значения элементов (0, 0) называются DC-компонентами, а остальные элементы — AC-компонентами.

На пятом этапе 64 элемента блока выстраиваются в ряд, к которому применяется кодирование длин серий. Чтобы сконцентрировать нули в конце ряда, сканирование блока выполняется зигзагом, как показано на рис. 7.8. В нашем примере в конце блока группируется 38 нулей. Эта строка нулей заменяется просто числом 38.

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Рис. 7.8. Порядок передачи квантованных значений

На последнем, шестом этапе к общему списку чисел применяется код Хаффмана (Huffman).

Алгоритм JPEG может показаться сложным, но это только потому, что он таким и является. Однако он широко применяется, так как позволяет сжимать фотографии в 20 и более раз. Для декодирования сжатого изображения нужно выполнить все те же операции в обратном порядке. В отличие от некоторых других алгоритмов, JPEG примерно симметричен: декодирование занимает столько же времени, сколько и кодирование.

Стандарт MPEG

Наконец мы подходим к ключевому вопросу мультимедиа: стандартам **MPEG** (Motion Pictures Experts Group — Экспертная группа по вопросам движущегося изображения). Эти стандарты, ставшие международными в 1993 году, описывают основные алгоритмы, используемые для сжатия видеофильмов. Первым законченным стандартом стал стандарт MPEG-1 (Международный стандарт 11172). Его целью было создание выходного потока данных качества бытового видеомагнитофона (352×240 для NTSC) на скорости 1,2 Мбит/с. Следом появился стандарт MPEG-2 (Международный стандарт 13818), разрабатывавшийся для сжатия видеофильмов качества широковещания до скорости потока от 4 до 6 Мбит/с, что позволяло передавать этот фильм в цифровом виде по стандартному телевизионному каналу NTSC или PAL.

В видеофильмах имеется избыточность двух типов: пространственная и временная. Чтобы использовать пространственную избыточность, можно просто кодировать каждый кадр отдельно алгоритмом JPEG. Дополнительного сжатия можно достичь, используя преимущество того факта, что последовательные кадры часто бывают почти идентичны (временная избыточность). Система **DV** (Digital Video — цифровое видео), применяющаяся в цифровых видеокамерах, использует только сжатие по алгоритму JPEG, так как кодирование должно производиться в режиме реального времени и значительно быстрее кодировать каждый кадр отдельно. Последствия такой схемы видны в табл. 7.1: хотя поток данных цифровой видеокамеры и ниже, чем у несжатого видео, он все же не настолько хорошо сжат,

как MPEG-2. (Чтобы это сравнение было честным, добавим, что DV-видеокамеры записывают значение сигнала яркости 8 битами, а каждое значение цветности — всего 2 битами).

В сценах, в которых камера и задний план неподвижны и один или два актера медленно двигаются, почти все пикселы в соседних кадрах будут идентичны. В этом случае простое вычитание каждого кадра из предыдущего и обработка разности алгоритмом JPEG даст достаточно хороший результат. Однако этот метод плохо подходит к сценам, в которых камера поворачивается или наезжает на снимаемый объект. Для таких сцен необходим какой-либо способ компенсировать это движение камеры. В этом и состоит основное отличие MPEG от JPEG.

Выход MPEG-2 состоит из кадров следующих типов:

1. I (Intracoded — автономные) — независимые неподвижные изображения, кодированные алгоритмом JPEG.
2. P (Predictive — предсказывающие) — содержащие разностную информацию, относительно предыдущего кадра.
3. B (Bidirectional — двунаправленные) — содержащие изменения относительно предыдущего и последующего кадров.

I-кадры представляют собой обычные неподвижные изображения, кодированные алгоритмом JPEG с использованием полного разрешения яркости и половинного разрешения для обоих сигналов цветности. I-кадры должны периодически появляться в выходном потоке по трем причинам. Во-первых, должна быть возможность просмотра фильма не с самого начала. Зритель, пропустивший первый кадр, не сможет декодировать все последующие кадры, если все кадры будут зависеть от предыдущих, и I-кадры не будут время от времени включаться в поток. Во-вторых, дальнейшее декодирование фильма станет невозможным в случае ошибки при передаче какого-либо кадра. В-третьих, наличие таких кадров существенно упростит индикацию во время быстрой перемотки вперед или назад. По этим причинам I-кадры включаются в выходной поток примерно один-два раза в секунду.

P-кадры, напротив, представляют собой разность между соседними кадрами. Они основаны на идее **макроблоков**, покрывающих 16×16 пикселов в пространстве яркости и 8×8 пикселов в пространстве цветности. При кодировании макроблока в предыдущем кадре ищется наиболее близкий к нему макроблок.

Пример использования макроблоков показан на рис. 7.9. Здесь мы видим три последовательных кадра с одинаковым задним планом, отличающихся только положением человека. Макроблоки, содержащие задний план, будут полностью совпадать друг с другом, но макроблоки, содержащие человека, будут смещаться на некоторую величину. Именно для этих макроблоков алгоритм должен найти максимально похожие макроблоки из предыдущего кадра.

Стандарт MPEG не указывает, как искать, насколько далеко искать или насколько хорошо должен подходить похожий макроблок. Все эти вопросы оставлены на усмотрение разработчика программы, реализующей стандарт MPEG. Например, макроблок можно искать в предыдущем кадре в текущей позиции с заданными смещениями по горизонтали и вертикали. Для каждой позиции может вычисляться количество совпадений матрицы яркости. Позиция с наибольшим значением совпадений будет объявляться победительницей при условии, что это значение превос-

ходит некое пороговое значение. В противном случае макроблок будет считаться не найденным. Возможны, конечно, и значительно более сложные алгоритмы.

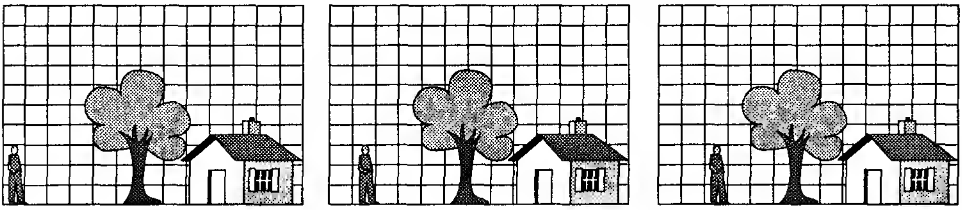


Рис. 7.9. Три последовательных кадра

Макроблок, для которого найден похожий на него макроблок в предыдущем кадре, кодируется в виде разности значений яркости и цветности. Затем матрицы разностей подвергаются дискретному косинусному преобразованию, квантованию, кодированию длин серий и кодированию Хаффмана так же, как это делает алгоритм JPEG. В выходном потоке макроблок представляется в виде вектора сдвига (насколько далеко сдвинулся макроблок по горизонтали и вертикали от положения в предыдущем кадре), за которым следует список чисел, кодированных по Хаффману. Если же в предыдущем кадре не нашлось подходящего макроблока, текущее значение кодируется алгоритмом JPEG, как в I-кадре.

В-кадры подобны Р-кадрам с той разницей, что позволяют привязывать макроблок либо к предыдущему, либо к следующему кадру. Такая дополнительная свобода позволяет достичь лучшей компенсации движения. Для декодирования В-кадров необходимо удерживать в памяти сразу три кадра: предыдущий, текущий и следующий. Для упрощения декодирования кадры должны присутствовать в потоке MPEG не в порядке их отображения, а в порядке зависимостей друг от друга. Это означает, что при передаче видео по сети необходима буферизация данных на машине пользователя, чтобы изменить порядок кадров для их правильного отображения. Поскольку порядок отображения кадров не совпадает с порядком их взаимозависимостей, для воспроизведения фильма задом наперед потребуется значительная буферизация и сложные алгоритмы.

Планирование процессов в мультимедийных системах

Операционные системы, поддерживающие мультимедиа, отличаются от обычных систем тремя параметрами: планированием процессов, файловой системой и дисковым планированием. Эти темы будут рассмотрены в следующих разделах.

Планирование однородных процессов

Простейшая разновидность видеосервера может поддерживать отображение фиксированного числа фильмов, использующих одинаковую скорость передачи данных, видеоразрешение, частоту кадров и другие параметры. При таких условиях

используется эффективный алгоритм планирования. Для каждого фильма создается отдельный процесс (или поток), чья работа заключается в чтении фильма с диска по кадру и передаче этого кадра пользователю. Поскольку все процессы одинаково важны, выполняют одинаковый объем работ на каждый кадр и блокируются, закончив обработку текущего кадра, то для управления этими процессами лучше всего использовать простой алгоритм поочередного планирования. К стандартному алгоритму следует лишь добавить механизм, следящий за временными параметрами, чтобы гарантировать, что каждый процесс работает с правильной частотой.

Один способ достижения правильного временного режима состоит в использовании управляющих часов, тикающих, скажем, 30 раз в секунду (для NTSC). При каждом тике все процессы запускаются последовательно в одном и том же порядке. Завершив свою работу, процесс обращается к системному вызову suspend, который освобождает центральный процессор до следующего тика часов. Затем все повторяется снова. Пока число процессов невелико и вся работа может быть выполнена за время длительности одного кадра, такой алгоритм планирования работает вполне успешно.

Общее планирование реального времени

К сожалению, простой алгоритм поочередного планирования редко может быть применен в реальной жизни. Количество пользователей меняется со временем, размеры кадров варьируются в широчайших пределах благодаря самой природе видеосжатия (I-кадры значительно крупнее P-кадров и B-кадров), кроме того, в различных фильмах может использоваться различное разрешение. В результате может оказаться, что разным процессам потребуется работа с разной частотой для выполнения различного объема работ и с различными сроками их окончания.

Такие соображения приводят к совершенно другой модели: нескольким процессам, соревнующимся за право использования центрального процессора, каждый со своим заданием и графиком его выполнения. В следующих моделях мы будем предполагать, что системе известна частота, с которой должен работать каждый процесс, объем работ, который ему предстоит выполнить, и ближайший срок выполнения очередной порции задания. (Дисковое планирование тоже важно, но мы обсудим этот вопрос позднее.) Планирование нескольких конкурирующих процессов, у некоторых (или у всех) из них есть жесткие сроки выполнения работ, называется **планированием реального времени**.

В качестве примера среды, в которой работает мультимедийный планировщик реального времени, рассмотрим три процесса, *A*, *B* и *C*, показанные на рис. 7.10. Процесс *A* запускается каждые 30 мс (приблизительная частота NTSC). Для обработки каждого кадра требуется около 10 мс времени центрального процессора. При отсутствии конкуренции он будет запускаться импульсами *A*1, *A*2, *A*3 и т. д., каждый из которых будет начинаться через 30 мс после предыдущего. Каждый 10-миллисекундный интервал работы центрального процессора обрабатывает один кадр и должен завершить работу к определенному моменту времени, то есть прежде, чем начнется обработка следующего кадра.

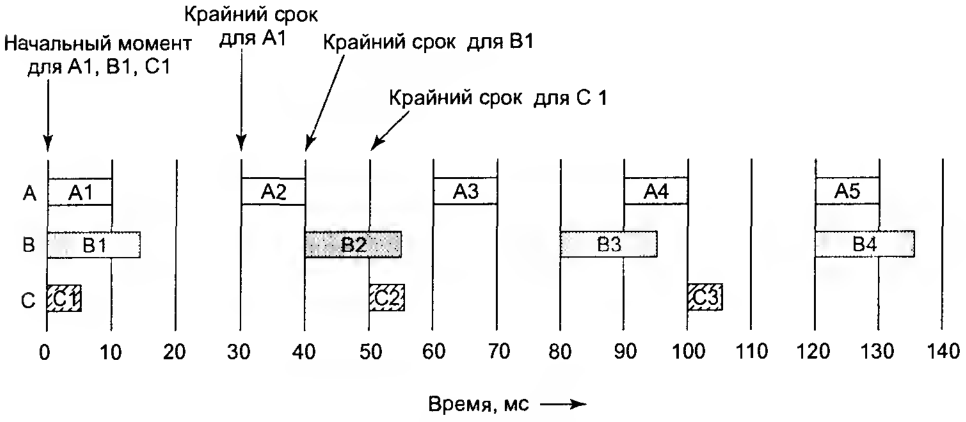


Рис. 7.10. Три периодических процесса воспроизведения фильмов. Частота кадров и время обработки кадра различны для каждого процесса

На рис. 7.10 также показаны два других процесса: *B* и *C*. Процесс *B* работает с частотой 25 кадров в секунду (например, PAL), а процесс *C* — с частотой 20 кадров в секунду (например, замедленный поток NTSC или PAL, предназначенный для пользователя, соединенного с видеосервером линией с низкой пропускной способностью). Время обработки каждого кадра показано на рисунке — 15 мс для процесса *B* и 5 мс для процесса *C*.

Проблема планирования состоит в том, как в данной ситуации планировать работу процессов *A*, *B* и *C* так, чтобы гарантировать выполнение ими требуемой работы в срок. Прежде чем начать знакомство с алгоритмом планирования, нам необходимо понять, возможно ли в принципе выполнение поставленной задачи, то есть является ли данный набор процессов планируемым вообще. Как говорилось в разделе «Планирование в системах реального времени» главы 2, если у процесса i период равен P_i и на обработку одного кадра этого процесса уходит C_i секунд работы процессора, то система будет планируемой только в том случае, если

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1,$$

где m — количество процессов, в данном случае 3. Обратите внимание, что C_i/P_i является просто частью процессорного времени, используемой процессом i. Например, на рис. 7.10 процесс *A* съедает 10/30 времени центрального процессора, процесс *B* съедает 15/40 времени центрального процессора, а процесс *C* съедает 5/50 времени центрального процессора. Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы, и, следовательно, система является планируемой.

До сих пор предполагалось наличие только одного процесса для каждого потока данных. В действительности их может быть два (и более), например один процесс для аудио и один для видео. Они могут работать с различными скоростями передачи данных и могут потреблять разное количество процессорного времени для каждого кадра.

В некоторых системах реального времени процессы являются прерываемыми, тогда как в других системах — нет. В мультимедийных системах процессы, как правило, могут прерываться. Это означает, что процесс, которому угрожает невыполнение задачи в срок, может прервать работающий процесс прежде, чем тот успеет закончить обработку своего кадра. Затем управление может быть возвращено прерванному процессу. Такое поведение процессов представляет собой многозадачность, о которой уже говорилось в предыдущих главах. Мы рассмотрим алгоритмы планирования реального времени с прерываниями, так как они не противоречат принципам мультимедийных систем и позволяют достичь лучших показателей производительности, чем алгоритмы без прерываний. Единственная забота состоит в том, что при заполнении буфера за короткие интервалы времени буфер должен быть заполнен в срок, чтобы его можно было отправить за одну операцию. В противном случае может возникнуть джиттер.

Алгоритмы реального времени могут быть статическими или динамическими. Статические алгоритмы заранее назначают каждому процессу фиксированный приоритет, после чего выполняют приоритетное планирование с переключениями. У динамических алгоритмов нет фиксированных приоритетов. Мы изучим примеры обоих типов алгоритмов.

Алгоритм планирования RMS

Классическим примером статического алгоритма планирования реального времени для прерываемых, периодических процессов является алгоритм **RMS** (Rate Monotonic Scheduling — планирование с приоритетом, пропорциональным частоте) [212]. Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода.
2. Ни один процесс не должен зависеть от любого другого процесса.
3. Каждому процессу требуется одинаковое процессорное время на каждом интервале.
4. У непериодических процессов нет жестких сроков.
5. Прерывание процесса происходит мгновенно, без накладных расходов.

Первые четыре требования разумны. Последнее, естественно, нет, но оно значительно упрощает модель системы. Алгоритм RMS работает, назначая каждому процессу фиксированный приоритет, равный частоте возникновения событий процесса. Например, процесс, который должен запускаться каждые 30 мс (33 раза в секунду), получает приоритет 33; процесс, который должен запускаться каждые 40 мс (25 раз в секунду), получает приоритет 25; а процесс, который должен запускаться каждые 50 мс (20 раз в секунду), получает приоритет 20. Таким образом, приоритеты пропорциональны частоте (количеству раз в секунду, которые запускается процесс). Отсюда и название алгоритма. Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим приоритетом, прерывая при необходимости работающий процесс. Авторы алгоритма доказали, что RMS является оптимальным решением в классе статических алгоритмов планирования.

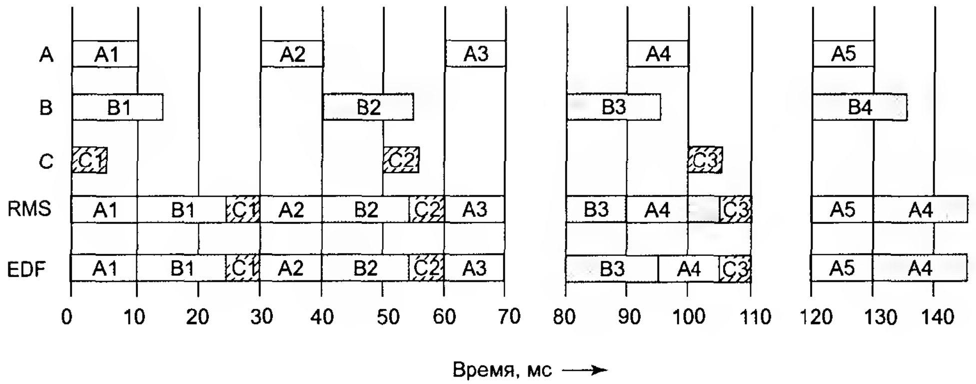


Рис. 7.11. Пример алгоритмов планирования реального времени RMS и EDF

На рис. 7.11 показана работа алгоритма планирования RMS в ситуации примера предыдущего рисунка. Процессам *A*, *B* и *C* назначены статические приоритеты 33, 25 и 20 соответственно; это означает, что когда процесс *A* желает работать, он работает, прерывая любой другой процесс, использующий центральный процессор в данный момент. Процесс *B* может прервать работу процесса *C*, но не *A*. Процесс *C* вынужден ждать, пока центральный процессор не освободится.

Изначально все три процесса готовы к работе (см. рис. 7.11). Выбирается процесс с максимальным приоритетом, то есть процесс *A*. Ему разрешается работать в течение 15 мс, требующихся процессу, чтобы полностью выполнить работу по передаче одного кадра (это показано в строке рисунка, помеченной RMS). Когда процесс *A* заканчивает свою работу, запускается процесс *B*, а затем процесс *C*. Вместе эти процессы потребляют 30 мс процессорного времени, поэтому, когда процесс *C* заканчивает свою работу, пора снова запускать процесс *A*. Этот цикл повторяется до тех пор, пока в момент времени $t = 70$ у системы начинается период простоя.

В момент времени $t = 80$ процесс *B* переходит в состояние готовности и запускается. Однако в момент времени $t = 90$ процесс *A*, обладающий более высоким приоритетом, также переходит в состояние готовности, поэтому он прерывает выполнение процесса *B* и работает, пока не закончит свою работу к моменту времени $t = 100$. В этом месте система должна выбрать между завершением процесса *B* и запуском процесса *C*. Выбирается, естественно, процесс *B* с более высоким приоритетом.

Алгоритм планирования EDF

Другим популярным алгоритмом планирования является алгоритм **EDF** (Earliest Deadline First — процесс с ближайшим сроком завершения в первую очередь). Алгоритм EDF представляет собой динамический алгоритм, в отличие от предыдущего алгоритма не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования центрального процессора. Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает

первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса.

Пример работы алгоритма EDF показан на рис 7.11. Вначале все процессы находятся в состоянии готовности. Они запускаются в порядке своих крайних сроков. Процесс *A* должен быть выполнен к моменту времени $t = 30$, процесс *B* должен закончить работу к моменту времени $t = 40$, и процесс *C* должен завершить работу к моменту времени $t = 50$. Таким образом, процесс *A* запускается первым. Вплоть до момента времени $t = 90$ выбор алгоритма EDF не отличается от RMS. В момент времени $t = 90$ процесс *A* снова переходит в состоянии готовности с тем же крайним сроком завершения $t = 120$, что и у процесса *B*. Планировщик имеет право выбрать любой из процессов, но поскольку с прерыванием процесса *B* не связано никаких накладных расходов, лучше предоставить возможность продолжать работу этому процессу.

Чтобы не дать сложиться ложному представлению, будто бы алгоритмы RMS и EDF всегда дают один и тот же результат, рассмотрим другой пример, показанный на рис. 7.12. В этом примере периоды процессов *A*, *B* и *C* те же, что и прежде, но теперь процессу *A* на передачу одного кадра требуется 15 мс процессорного времени вместо 10 мс. Тест планируемости дает коэффициент использования процессора, равный 0,500 + 0,375 + 0,100 = 0,975. В запасе остается всего лишь 2,5 % времени центрального процессора, но теоретически коэффициент использования процессора не превышает допустимого предела, поэтому для данного случая должен существовать метод планирования.

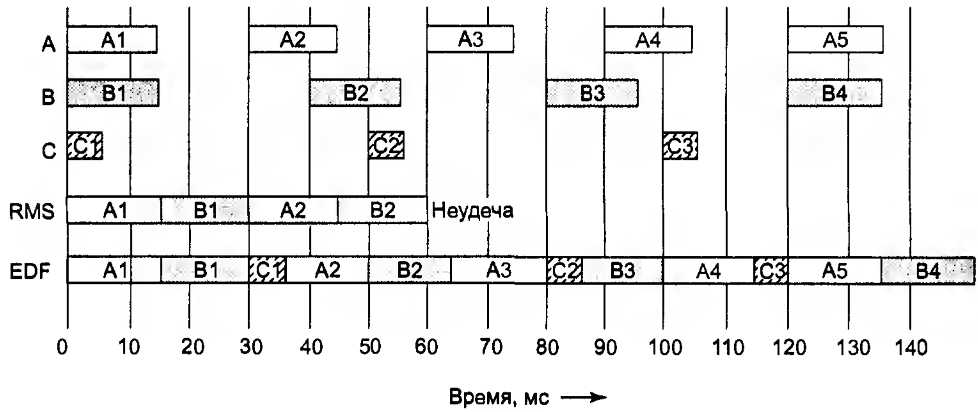


Рис. 7.12. Другой пример алгоритмов планирования реального времени RMS и EDF

В алгоритме RMS приоритеты трех процессов по-прежнему равны 33, 25 и 20, так как на них влияют только периоды, а не времена работы. На этот раз процесс *B*1 завершает работу к моменту времени $t = 30$, однако в этот момент процесс *A* снова приходит в состояние готовности. К тому моменту, когда он заканчивает свою работу ($t = 45$), процесс *B* также уже снова готов работать, и поскольку у него приори-

тет выше, чем у процесса *C*, то запускается процесс *B*, а процесс *C* пропускает свой критический срок. Таким образом, алгоритм RMS терпит фиаско.

Теперь посмотрим, как алгоритм EDF справляется с данным случаем. В момент времени $t = 30$ между процессами *A*2 и *C*1 возникает спор. Поскольку срок выполнения процесса *C*1 равен 50 мс, а срок выполнения процесса *A*2 равен 60 мс, планировщик выбирает процесс *C*. Этим данный алгоритм отличается от RMS, в котором побеждает процесс *A*, как обладающий более высоким приоритетом.

В момент времени $t = 90$ процесс *A* приходит в состояние готовности в четвертый раз. Предельный срок процесса *A* такой же, что и текущего процесса (120 мс), поэтому у планировщика появляется выбор: прервать работу процесса *B* или нет. Поскольку необходимости прерывать процесс *B* нет, то процессу *B*3 разрешается продолжать работу.

В данном примере центральный процессор занят на все 100 % до момента времени $t = 150$. Однако в конце концов перерыв в его работе наступает, так как центральный процессор занят всего на 97,5 %. Поскольку время и начала и окончания работ во всех случаях кратно 5 мс, промежуток простоя центрального процессора будет составлять 5 мс. Чтобы относительное время простоя было равно 2,5 %, 5-миллисекундный интервал бездействия должен происходить каждые 200 мс, что не поместилось на рис. 7.12.

Интересно, почему потерпел неудачу алгоритм RMS. В основном дело в том, что использование статических приоритетов работает только при не слишком высокой загруженности центрального процессора. Как показали Лю и Лэйланд [212], алгоритм RMS гарантированно работает в любой системе периодических процессов при условии

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le m(2^{1/m} - 1).$$

Для 3, 4, 5, 10, 20 и 100 процессов максимальная разрешенная загруженность процессора составляет 0,780, 0,757, 0,743, 0,718, 0,705 и 0,696. При $m \to \infty$ значение максимальной загруженности процессора асимптотически стремится к ln 2. Другими словами, Лю и Лэйланд доказали, что для трех процессов алгоритм RMS всегда работает при коэффициенте загруженности центрального процессора не выше 0,780. В нашем первом примере коэффициент загруженности составлял 0,808, и алгоритм RMS работал, но нам просто повезло. С другими периодами и временем работы процессов данный алгоритм может потерпеть неудачу при такой загруженности центрального процессора. Во втором примере коэффициент загруженности центрального процессора оказался настолько высок (0,975), что на возможность работы алгоритма RMS просто не было надежды.

Алгоритм EDF, напротив, всегда работает с любым набором процессов, для которого возможно планирование. Коэффициент загруженности центрального процессора для алгоритма EDF может достигать 100 %. Платой за это является использование более сложного алгоритма. Таким образом, на реальном видеосервере при загруженности центрального процессора ниже предела RMS может использоваться алгоритм RMS. В противном случае должен применяться алгоритм EDF.

Парадигмы мультимедийной файловой системы

Описав планирование процессов в мультимедийных системах, продолжим наше изучение мультимедийных файловых систем. В этих файловых системах применяются парадигмы, отличные от используемых в традиционных файловых системах. Сначала мы рассмотрим традиционный файловый ввод-вывод, затем обратим внимание на то, как организованы мультимедийные файловые серверы. Для доступа к файлу процесс сначала обращается к системному вызову open. Если эта операция проходит успешно, процессу возвращается нечто вроде маркера, называемого дескриптором или описателем файла. С этого момента процесс может обращаться к системному вызову read, указывая на входе полученный маркер, адрес буфера и счетчик байтов в качестве параметров. При этом операционная система возвращает в буфер требуемые данные. Пока процесс не завершил свою работу, он может издавать дополнительные системные вызовы read, а затем процесс должен обратиться к системному вызову close, чтобы закрыть файл и вернуть ресурсы системе.

Для мультимедиа эта модель работает плохо, так как не отвечает требованиям реального времени. Особенно плохо подходит такая схема для отображения мультимедийных файлов, поступающих с видеосервера. Одна проблема состоит в том, что пользователь должен обращаться к системному вызову read через точные интервалы времени. Вторая проблема заключается в том, что видеосервер должен предоставлять данные без задержки, что сложно в ситуации, когда запросы поступают стихийно, а ресурсы не резервируются заранее.

Для решения этих проблем мультимедийными файловыми серверами используется совершенно другая парадигма: они действуют подобно кассетным видеомагнитофонам. Для чтения мультимедийного файла пользовательский процесс обращается к системному вызову start, указывая файл, который следует прочитать, а также другие параметры, например какую звуковую дорожку и субтитры использовать. Затем видеосервер начинает посылать кадры с требуемой частотой. Принять их и обработать является задачей пользователя. Если пользователю наскучит фильм, он может остановить поступающий поток при помощи системного вызова stop. Файловые серверы, работающие по такой схеме, часто называют **пуш-серверами** (push servers, то есть серверы, «толкающие» данные пользователю). Традиционные серверы, у которых пользователь должен запрашивать данные поблочно, в цикле обращаясь к системному вызову read, называют **пул-серверами** (pull servers). Различие между этими двумя моделями проиллюстрировано на рис. 7.13.

Функции управления видеомагнитофоном

Большинство видеосерверов также реализуют стандартные управляющие функции видеомагнитофонов, включая паузу, быструю перемотку вперед и назад. Пауза реализуется просто. Пользователь посылает видеосерверу сообщение с просьбой остановить поток данных. Все, что требуется от видеосервера, — это запомнить, с какого кадра продолжать передачу. Когда пользователь просит сервер возобновить передачу, тот просто продолжает с места, на котором был остановлен.

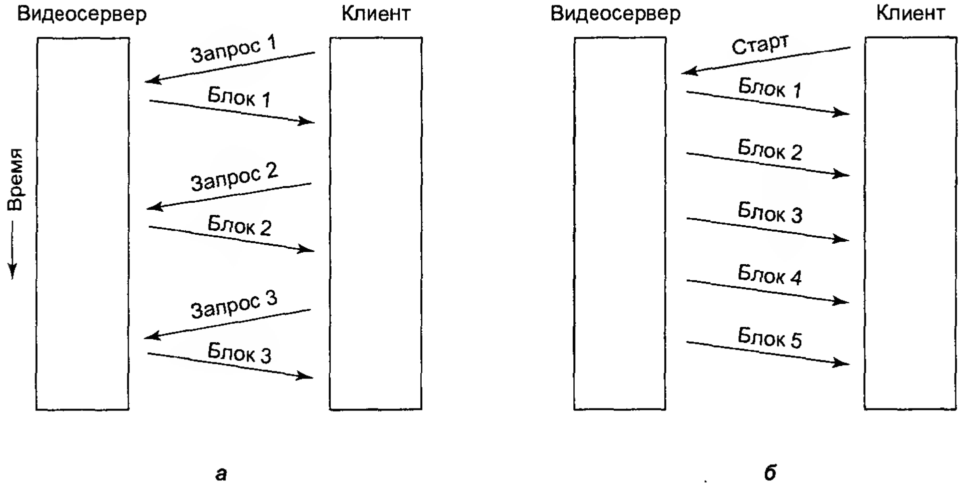


Рис. 7.13. Пул-сервер (*а*); пуш-сервер (*б*)

Однако здесь имеется одна проблема. Для достижения приемлемой производительности сервер может зарезервировать для каждого исходящего потока такие ресурсы, как пропускную способность диска и буферы памяти. Продолжать удерживать эти ресурсы, в то время как фильм стоит на паузе, неэффективно. Сервер может освобождать ресурсы на время паузы, но при этом возникает опасность, что когда пользователь захочет продолжить просмотр фильма, эти ресурсы уже будут заняты.

Перемотка на начало фильма также очень проста и обходится без каких-либо проблем. Все, что нужно сделать серверу, — это просто начать передавать фильм снова с кадра номер 0. Что может быть проще? Однако быстрая перемотка вперед или назад (то есть с воспроизведением при перемотке) значительно сложнее. Если бы не временное сжатие, применяемое в MPEG, для быстрой перемотки вперед с удесятеренной скоростью было бы достаточно просто отображать каждый десятый кадр. Для ускорения в 20 раз нужно было бы отображать каждый 20-й кадр. В самом деле, при отсутствии сжатия быстрая перемотка вперед и назад не представляют трудностей. Для быстрой перемотки вперед с ускорением в k раз достаточно просто отображать каждый k-й кадр. Для быстрой перемотки назад в k раз быстрее нормальной скорости нужно выполнять те же действия в противоположном направлении. Этот подход в равной мере годится для пул-серверов и пуш-серверов.

Все становится значительно сложнее из-за сжатия. В случае видеокамеры, хранящей каждый кадр в сжатом виде, независимо от других кадров, такая стратегия может использоваться при условии, что каждый кадр может быть быстро найден. Поскольку каждый кадр сжимается по-разному, в зависимости от его содержимого, каждый кадр в сжатом виде имеет различный размер, поэтому пропуск k кадров не может быть выполнен простым умножением размера кадра на k. Кроме того, звук сжимается отдельно от изображения, поэтому для каждого отображаемого в ускоренном режиме видеокадра должен быть также обнаружен правильный аудиокадр (если только звук не отключается при быстрой перемотке). Таким

образом, для быстрой перемотки файла формата Digital Video требуется индекс, позволяющий быстро находить нужные кадры, но по крайней мере теоретически такая задача вполне выполнима.

С файлами в формате MPEG такая схема не работает даже в теории, так как в данном стандарте используются I-, P- и B-кадры. Пропуск *k* кадров может привести к тому, что программа получит следующим P-кадр, для правильной расшифровки которого необходим пропущенный I-кадр. Без базового кадра относительные величины, хранящиеся в P-кадре, бесполезны. Таким образом, стандарт MPEG требует последовательного воспроизведения файла.

Для решения этой проблемы можно попытаться действительно воспроизводить файл последовательно с удесятеренной скоростью. Однако для этого требуется считывание данных с диска в десять раз быстрее. Затем сервер может попытаться распаковать кадры (чего он не делает при нормальной работе), определить, который кадр требуется, и снова запаковать каждый 10-й кадр как I-кадр. Однако это явится колоссальной нагрузкой на сервер. Кроме того, от сервера требуется понимание формата сжатия данных, что в нормальных условиях ему не нужно.

Альтернативное решение состоит в передаче всех данных по сети пользователю. Однако это решение потребует работы сети на десятикратной скорости, что, возможно, и достижимо, но не слишком просто, учитывая и без того высокие скорости, на которых приходится работать мультимедийным сетям в нормальной ситуации.

В общем, простого решения этой проблемы не существует. Единственная осуществимая стратегия заключается в заблаговременном создании специального файла, содержащего, скажем, каждый 10-й кадр, и сжатии этого файла с помощью обычного алгоритма MPEG. Этот файл был показан на рис. 7.2 как «быстрая перемотка вперед». Для переключения в режим быстрой перемотки вперед серверу нужно всего лишь определить, в каком месте этого файла находится в данный момент текущий указатель просматриваемого фильма. Например, если номер текущего кадра равен 48 210, а файл быстрой перемотки вперед ускоряет просмотр в 10 раз, то сервер должен найти в этом файле кадр 4821 и начать его воспроизведение с нормальной скоростью. Конечно, этот кадр может оказаться P- или B-кадром, но клиентский процесс декодирования может просто пропустить их, пока не наткнется на I-кадр. Обратная перемотка реализуется аналогично при помощи второго специально приготовленного файла.

Когда пользователь переключается обратно в режим нормального воспроизведения, сервер должен произвести обратный перерасчет номеров кадров и переключиться на нормальный файл.

Хотя наличие этих двух специальных файлов позволяет реализовать быструю перемотку вперед и назад, у такого подхода имеются некоторые недостатки. Во-первых, для хранения дополнительных файлов требуется дополнительное дисковое пространство. Во-вторых, при этом быстрая перемотка вперед и назад может выполняться только на скоростях, соответствующих скоростям этих специальных файлов. В-третьих, для переключения из одного режима в другой и обратно требуется дополнительное усложнение серверной части.

«Почти видео по заказу»

Если *k* пользователей одновременно смотрят один и тот же фильм, на сервер ложится практически такая же нагрузка, как если бы они смотрели *k* различных фильмов. Однако при небольшом изменении существующей модели можно достичь большого выигрыша в производительности. Видео по заказу предоставляет возможность пользователям начать смотреть фильм в произвольный момент. Поэтому, если 100 пользователей начнут просмотр нового фильма около 8 часов вечера, есть шанс, что никто из них не подаст запрос одновременно, и, следовательно, они не смогут совместно воспользоваться одним потоком. Можно существенно оптимизировать данную схему, разрешив пользователям начинать просмотр фильма не в любой момент, а с интервалом в 5 мин. Таким образом, если пользователь захочет смотреть фильм в 20:02, показ фильма для него начнется в 20:05.

Выигрыш такой стратегии заключается в том, что для 2-часового фильма потребуется всего 24 потока данных, независимо от того, сколько зрителей будут смотреть этот фильм. Как показано на рис. 7.14, первый поток начинается в 20:00. В 20:05, когда первый поток передает кадр номер 9000, начинается передача потока 2. В 20:10, когда первый поток передает кадр номер 18 000, а второй поток передает кадр номер 9000, начинается передача потока 3 и т. д. до потока 24, начинающегося в 21:55. В 22:00 поток 1 завершается и начинается снова с кадра 0. Эта схема называется **«почти видео по заказу»**, так как показ фильма не начинается сразу после поступления заказа, а некоторое время спустя.

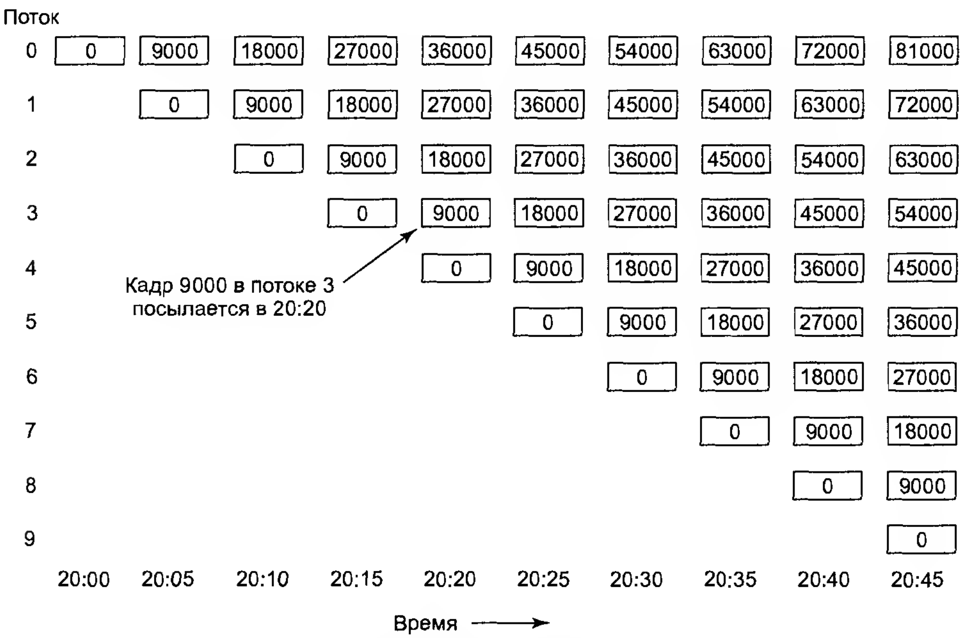


Рис. 7.14. В схеме «почти видео по заказу» новый поток запускается через равные интервалы времени, например 5 мин (9000 кадров)

Ключевым параметром в данной схеме является частота запусков потоков. Если потоки запускать через каждые 2 мин, то для демонстрации 2-часового фильма потребуется 60 потоков, зато максимальное время ожидания начала просмотра будет всего 2 мин. Оператор должен принять решение об этом параметре, так как увеличение интервала между потоками позволит увеличить эффективность системы, увеличив количество одновременно демонстрируемых фильмов. Альтернативная стратегия состоит в том, что фильм запускается сразу же, но такая служба будет обходиться клиенту дороже.

В каком-то смысле видео по заказу напоминает такси: вы звоните, и оно приезжает. «Почти видео по заказу» больше напоминает автобус, ходящий по расписанию, вам нужно всего лишь подождать следующего. Но общественный транспорт имеет смысл только в том случае, когда требуется перевезти большое количество людей. Автобус, проходящий по центральному Манхеттену каждые 5 мин, может рассчитывать на нескольких пассажиров. Автобус, двигающийся по второстепенным дорогам Вайоминга, может быть пуст почти всю дорогу. Аналогично, показ последнего фильма Стивена Спилберга может привлечь достаточное количество зрителей, чтобы запускать новый поток с фильмом каждые 5 мин, но для «Унесенных ветром», возможно, лучше подойдет вариант индивидуального заказа.

В системе «почти видео по заказу» пользователи не имеют возможности управлять демонстрацией фильма. Пользователь не может поставить фильм на паузу, чтобы прогуляться на кухню. В данном случае по возвращении из кухни пользователю придется переключиться на другой канал, показывающий этот фильм с опозданием на 5 или 10 мин.

В принципе такая система видео может быть частично интерактивной, то есть не просто запускать один фильм с интервалом в 5 мин, а делать это по просьбам зрителей. Раз в 5 мин такая система проверяет, на какие фильмы поступают заказы, и показывает именно их. При таком подходе фильм может начаться по заказу, например, в 20:00, 20:10, 20:15 и 20:25, но не в промежутках между этими временами. В результате потоки, у которых нет зрителей, не занимают линию, экономя также пропускную способность диска и память видеосервера. С другой стороны, атака на холодильник в такой ситуации становится рискованным предприятием, так как нет никакой гарантии, что вы сможете переключиться на другой поток, отстающий на 5 мин от только что просматриваемого вами. Конечно, оператор может предоставить пользователям список всех передаваемых в данный момент потоков, однако многие телезрители полагают, что у пультов дистанционного управления их телевизоров и так слишком много кнопок, поэтому они вряд ли обрадуются появлению еще одной.

«Почти видео по заказу» с функциями видеомагнитофона

Идеальной комбинацией было бы «почти видео по заказу» (с его экономичностью) плюс полный комплект возможностей видеомагнитофона для каждого индивидуального пользователя (для удобства зрителя). При небольших изменениях такая модель осуществима. Ниже будет дано слегка упрощенное описание одного из способов достижения данной цели [2].

Мы начнем со стандартной схемы «почти видео по заказу» (см. рис. 7.14). Однако добавим к ней требование локального буферирования на машине клиента предыдущих ΔT мин и последующих ΔT мин фильма. Буферизация предыдущих ΔT мин не представляет трудности: нужно просто сохранять поступающие данные после их отображения. Буферизация последующих ΔT мин фильма несколько сложнее, но также осуществима, если машина клиента способна принимать сразу два потока.

Проиллюстрируем на примере один вариант установки буфера. Если пользователь начинает просмотр в 8:15, клиентская машина читает и отображает поток 8:15, который передает кадр 0. Параллельно она читает и сохраняет на диске поток, начавшийся в 8:10, находящийся в данный момент на 5-минутной отметке (то есть кадр 9000). В 8:20 кадры от 0 до 17 999 сохранены на диске, а пользователь собирается смотреть кадр 9000. С этого момента поток 8:15 более не нужен, так как буфер заполнен потоком 8:10 (который передает кадр 18 000), и клиентская машина может продолжать отображение из середины буфера (кадр 9000). При чтении каждого кадра один новый кадр добавляется в буфер с одного конца, а с другого конца буфера один кадр удаляется. Кадр, отображаемый в данный момент, называемый **точкой воспроизведения**, всегда находится в середине буфера. Ситуация 75-й минуты фильма показана на рис. 7.15, *а*. На этом рисунке все кадры от 70-й до 80-й минуты фильма находятся в буфере. При скорости передачи данных 4 Мбит/с 10-минутный буфер должен иметь размер в 300 млн байт. При сегодняшних ценах такой буфер, конечно, может храниться на диске и, возможно, в ОЗУ. Если требуется ОЗУ, но буфер в 300 млн байт кажется чрезмерным, можно использовать буфер меньших размеров.

Теперь предположим, что пользователь решает перемотать фильм вперед или назад. Пока точка воспроизведения остается в интервале 70–80 мин, процесс отображения может получать данные из буфера. Однако если точка воспроизведения переместится за пределы этого интервала, возникает проблема. Решение заключается в переключении на персональный (то есть видео по заказу) канал обслуживания пользователя. Для быстрого перемещения в любом направлении могут использоваться обсуждавшиеся ранее методы.

Как правило, в некоторый момент пользователь прекращает быструю перемотку и решает продолжить просмотр фильма в нормальном режиме. В этот момент можно подумать о том, чтобы переместить пользователя на один из потоков «почти видео по заказу», чтобы освободить более дорогой индивидуальный канал. Предположим, например, что пользователь решает вернуться на 12-минутную отметку (рис. 7.15, *б*). Эта отметка находится далеко за пределами буфера, поэтому для продолжения показа фильма буфер оказывается бесполезен. Более того, поскольку пользователь (мгновенно) переключился на 12-ю минуту с 75-й минуты фильма, то имеются потоки, демонстрирующие этот фильм на 5-й, 10-й, 15-й и 20-й минуте, но ни одного на 12-й.

Решение состоит в продолжении просмотра индивидуального канала, но с одновременным заполнением буфера из потока, показывающего в данный момент 15-ю минуту фильма. Ситуация через 3 мин после этого показана на рис. 7.15, *в*. Теперь точка воспроизведения находится на отметке 15 мин, в буфере содержатся данные фильма с 15-й по 18-ю минуту, а потоки «почти видео по заказу», среди прочих, демонстрируют 8-ю, 13-ю, 18-ю и 23-ю минуты. В этот момент индивидуальный поток может быть освобожден, а отображение может далее получать дан-

ные из буфера. Буфер продолжает пополняться из потока, передающего в этот момент 18-ю минуту. Еще через минуту точка воспроизведения перемещается на 16-ю минуту фильма, в буфере к этому моменту содержатся минуты от 15-й по 19-ю, а из потока в буфер поступает 19-я минута (рис. 7.15, *г*).

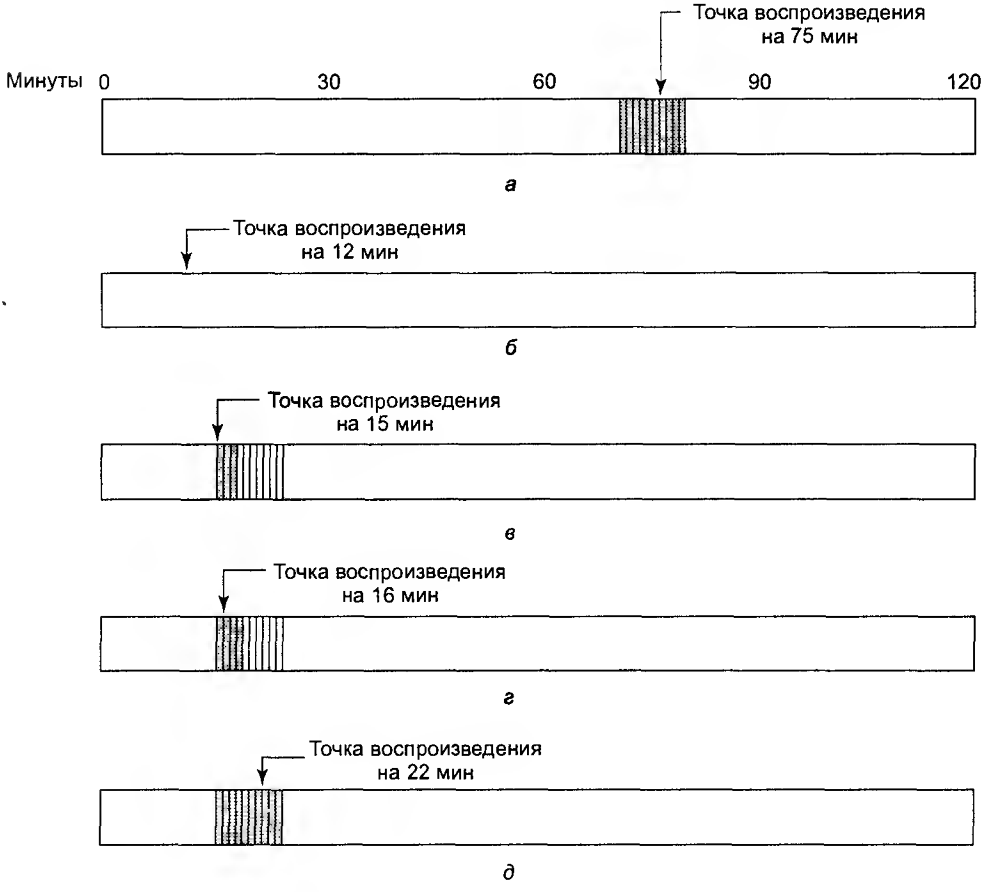


Рис. 7.15. Начальная ситуация (*а*); после обратной перемотки на 12 мин (*б*); после 3-минутного ожидания (*в*); буфер начинает заполняться (*г*); буфер полон (*д*)

Еще через 6 мин буфер заполняется, при этом точка воспроизведения находится на 22-й минуте фильма. Теперь она не в середине буфера, но при необходимости и этот вопрос может быть решен.

Размещение файла

Размер мультимедийных файлов очень велик. Мультимедийные файлы, как правило, записываются всего один раз, но много раз считываются. Доступ к мультимедийным файлам чаще всего последовательный. Их воспроизведение также долж-

но удовлетворять строгим критериям качества обслуживания. Все эти свойства мультимедийных файлов предполагают использование компоновки файловой системы, отличной от используемой в традиционных операционных системах. Некоторые из этих вопросов будут рассмотрены ниже, сначала мы рассмотрим однодисковый вариант хранения файла, а затем многодисковый.

Размещение файла на одном диске

Самое важное требование заключается в том, чтобы данные могли передаваться в сеть или выходное устройство с требуемой скоростью и без джиттера. По этой причине выполнение нескольких операций поиска цилиндра во время считывания кадра крайне нежелательно. Одним из способов устранения излишних перемещений блока головок на сервере является использование непрерывных файлов. Поскольку файлы с фильмами не изменяются после записи их на сервер, такая схема является реализуемой и работоспособной.

Однако имеется одна проблема, связанная с тем, что в таком файле одновременно должны храниться видео- и аудиоданные, а также текстовая информация (см. рис. 7.2). Даже если вся эта информация хранится в отдельных непрерывных файлах, потребуются операции поиска цилиндра при переключении с видеофайла на аудиофайл, а с него на текстовый файл. В результате был разработан другой вариант хранения, с чередованием видео-, аудио- и текстовой информации в одном файле (рис. 7.16). При этом сам файл остается непрерывным. В таком файле сразу за первым видеокадром следуют различные звуковые дорожки к первому кадру, а затем различные текстовые данные к этому же кадру. В зависимости от того, насколько много содержится в файле звуковых и текстовых дорожек, может оказаться проще прочитать их все за одну дисковую операцию, а пользователю передать только требующуюся часть этой информации.

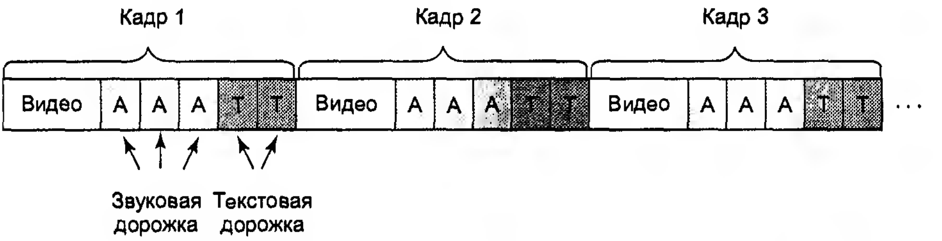


Рис. 7.16. Чередование видео-, аудио- и текстовой информации в одном непрерывном файле

Подобная организация требует дополнительных операций ввода-вывода для чтения ненужных звуковых дорожек и текста и дополнительного буферного пространства в памяти для их хранения. Однако такой подход устраняет все лишние операции по перемещению блока головок (в однопользовательской системе), а также накладные расходы по учету расположения кадров на диске, так как все кадры располагаются в непрерывном файле друг за другом. Правда, при таком размещении файла становится невозможным произвольный доступ к файлу, но если он не требуется, то такая потеря не страшна. Также без дополнительных структур

данных и усложнения алгоритмов невозможной становится быстрая перемотка вперед и назад.

Преимущество размещения всего фильма в виде одного непрерывного файла теряется на видеосервере, обслуживающем одновременно несколько выходных потоков, так как после чтения одного кадра фильма, прежде чем диск получит запрос на чтение следующего кадра, серверу потребуются кадры из других фильмов. Кроме того, в системе, в которой фильмы не только считываются, но и записываются (то есть используемой для производства или редактирования видеофильмов), использование огромных непрерывных файлов неудобно и трудно выполнимо.

Две альтернативные стратегии организации файлов

Результатом описанных выше наблюдений явилось создание двух альтернативных стратегий организации мультимедийных файлов. Первая из них, представляющая собой использование дисковых блоков небольшого размера, показана на рис. 7.17. В данной схеме размер блоков диска выбирается существенно меньшим, чем средний размер кадра, даже P-кадров и B-кадров. Для формата MPEG-2 со скоростью 4 Мбит/с при 30 кадрах в секунду средний кадр имеет размер около 16 Кбайт, поэтому в данном варианте хорошо подойдут дисковые блоки размером 1 Кбайт или 2 Кбайт. Идея заключается в том, чтобы получить структуру данных, а именно индекс кадров для каждого фильма, с указателями на начало каждого кадра. Каждый кадр содержит всю видео-, аудио- и текстовую информацию для этого кадра в виде непрерывной последовательности дисковых блоков (см. рис. 7.17). Таким образом, чтобы прочитать кадр *k*, нужно найти в индексе *k*-й элемент, а затем считать весь кадр за одну дисковую операцию. Поскольку различные кадры имеют различный размер, этот размер (в блоках) должен содержаться в индексе. Но при 1-килобайтных дисковых блоках 8-битового поля будет вполне достаточно, чтобы поддерживать кадры размером до 255 Кбайт, а этого достаточно для несжатого NTSC, даже при большом количестве звуковых дорожек.

Другой способ хранения фильма показан на рис. 7.17, *б*. В этом случае используются дисковые блоки большого размера (например, 256 Кбайт), и в каждый такой блок помещается несколько кадров. Индекс также нужен, но теперь это уже не индекс кадров, а индекс блоков. В принципе данный индекс почти не отличается от i-узла (см. рис. 6.12), возможно, с добавлением информации, сообщающей номер кадра, с которого начинается блок, что позволяет быстро находить нужные кадры. В общем случае целое число кадров не поместится в один блок. Существует два варианта решения данной проблемы.

Во-первых, оставшуюся часть блока можно оставлять неиспользуемой (рис. 7.17, *б*). При этом часть дискового пространства будет теряться, что представляет собой внутреннюю фрагментацию (то же, что и в системе виртуальной памяти со страницами фиксированного размера). С другой стороны, при таком подходе никогда не придется посреди чтения кадра выполнять операцию поиска дискового цилиндра.

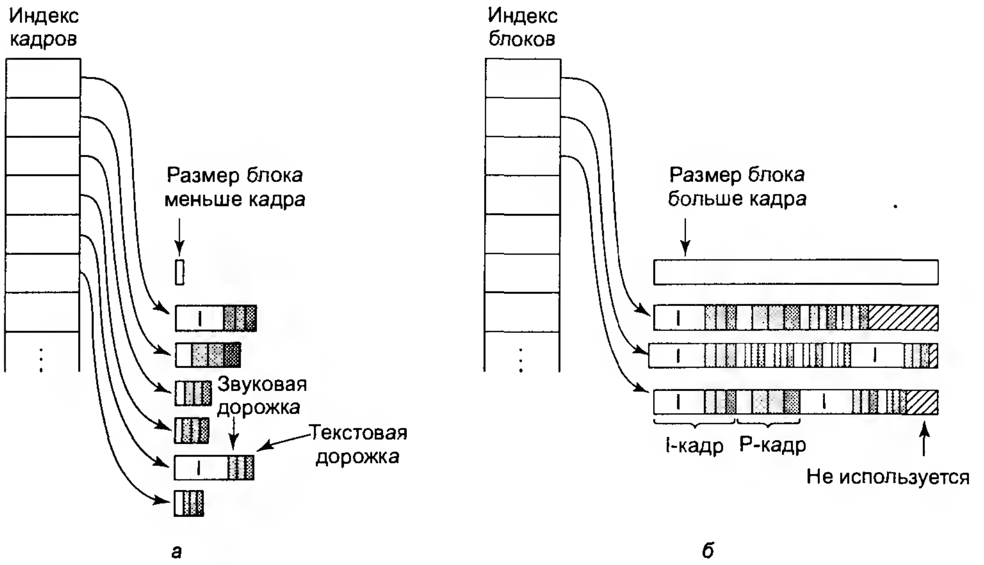


Рис. 7.17. Сегментированное хранение фильма: маленькие дисковые блоки (*а*); большие дисковые блоки (*б*)

Во-вторых, оставшаяся часть блока может заполняться частью кадра. В результате при чтении кадра может потребоваться перемещение блока головок, что снизит производительность, но зато позволит сэкономить некоторую часть дискового пространства, так как внутренняя фрагментация будет устранена.

В первом варианте хранения файлов на диске, состоящем из небольших блоков (см. рис. 7.17, *а*), также терялась определенная часть дискового пространства, потому что часть последнего блока в каждом кадре тоже не использовалась. При килобайтных дисковых блоках 2-часовой фильм в формате NTSC состоит из 216 000 кадров. При этом потери дискового пространства будут составлять всего 108 Кбайт из 3,6 Гбайт. Для ситуации на рис. 7.17, *б* потери дискового пространства сосчитать сложнее. Можно лишь сказать, что они будут значительно большими, так как время от времени в конце блока неиспользованными будут оставаться фрагменты блока размером порядка 100 Кбайт, в случае когда следующий I-кадр окажется больше этого размера.

С другой стороны, блоковый индекс занимает существенно меньше места, нежели кадровый индекс. При 256-килобайтных дисковых блоках и среднем размере кадров в 16 Кбайт в блок будет помещаться около 16 кадров, поэтому для фильма из 216 000 кадров потребуется всего около 13 500 элементов блочного индекса, против 216 000 для кадрового индекса. Для повышения производительности в обоих случаях индекс должен содержать все кадры или блоки (то есть не должны применяться косвенные блоки, как в UNIX). Таким образом, хранение в оперативной памяти 13 500 8-байтовых элементов (4 байта для дискового адреса, 1 байт для размера кадра и 3 байта для номера начального кадра) против 216 000 5-байтовых элементов (только дисковые адреса и размер) позволит сэкономить почти 1 Мбайт ОЗУ при воспроизведении фильма.

В результате вышеописанных соображений возникают следующие области возможных компромиссных решений:

1. Кадровый индекс: большие расходы ОЗУ при воспроизведении фильма; меньшие потери дискового пространства.
2. Блочный индекс (без расщепления кадров по блокам): меньшие потребности в ОЗУ; большие потери дискового пространства.
3. Блочный индекс (с расщеплением кадров по блокам): меньшие потребности в ОЗУ; нет потерь дискового пространства; лишние перемещения головок.

Компромиссные параметры представляют собой использование ОЗУ при воспроизведении фильма, постоянные потери дискового пространства и снижение производительности в результате лишних операций перемещения головок во время воспроизведения. Однако существуют различные методы решения данных проблем. Использование ОЗУ может быть уменьшено при помощи выгрузки на диск части таблицы кадров. Перемещения головок при передаче кадра можно скрыть с помощью достаточного буферирования, хотя для этого потребуются дополнительная оперативная память и, возможно, дополнительные операции копирования. В хорошо продуманной системе следует тщательно проанализировать все эти факторы и сделать правильный выбор для конкретного приложения.

Еще один фактор связан с тем, что использование способа хранения файла, показанного на рис. 7.17, *а*, усложняется необходимостью поиска непрерывной последовательности свободных блоков нужной длины. В идеале такая последовательность блоков не должна пересекать границу дорожек диска, хотя при использовании перекоса головок потери будут невелики. Тем не менее пересечения границы цилиндров следует избегать. Такое требование означает, что свободное дисковое пространство должно быть организовано скорее в виде списка незанятых участков диска переменного размера, нежели в виде простого списка свободных блоков или битового массива (оба эти метода могут использоваться в варианте на рис. 7.17, *б*).

Следует также отметить, что во всех случаях необходимо размещать блоки или кадры фильма по возможности близко друг от друга, например в пределах нескольких дисковых цилиндров. При таком расположении потребуется меньшее количество операций перемещений головок, в результате чего у сервера останется больше времени для других задач (не являющихся задачами реального времени) или для поддержки дополнительных видеопотоков. Подобное требование может быть удовлетворено при помощи разбиения диска на группы цилиндров, для каждой из которых будет поддерживаться отдельный список или битовый массив свободных блоков. Например, при учете свободных непрерывных участков диска можно поддерживать один список для свободных участков диска размером в 1 Кбайт, один для 2-килобайтных участков, еще один для участков размером от 3 до 4 Кбайт, следующий — для участков размером от 5 до 8 Кбайт и т. д. При этом программа сможет легко найти свободный участок диска нужного размера.

Другое различие между этими двумя подходами заключается в буферизации. При использовании блоков маленького размера при каждой операции чтения считывается ровно один кадр. Соответственно, простая стратегия двойной буфериза-

ции хорошо работает: один буфер для воспроизведения текущего кадра и один для считывания следующего. При использовании фиксированных буферов каждый буфер должен быть достаточно большим, чтобы вместить самый большой I-кадр. С другой стороны, при динамическом выделении буфера для чтения каждого кадра и размере кадра, известном до операции чтения, для Р-кадров и В-кадров может выделяться буфер меньшего размера.

При блоках большого размера требуется более сложная стратегия, так как каждый большой блок содержит несколько кадров, возможно, включая фрагменты кадров в конце каждого блока (в зависимости от выбранного варианта). Если для отображения или передачи кадров требуется, чтобы кадры были непрерывными, они должны копироваться, но копирование представляет собой дорогую операцию, которой следует по возможности избегать. Если непрерывность не требуется, то кадры, расположенные в двух блоках, могут передаваться по сети или на устройство воспроизведения двумя порциями.

При использовании больших блоков двойная буферизация может также применяться, но резервирование буферов для двух больших блоков приведет к большим расходам оперативной памяти. Один способ решения этой проблемы заключается в использовании (для каждого потока) циклического буфера передачи, размерами слегка превосходящего размеры дискового блока, из которого информация передается на дисплей или в сеть. Когда количество полезной информации в буфере снижается до некой пороговой величины, с диска считывается следующий большой блок, его содержимое копируется в буфер передачи, а большой буфер блока возвращается в общий пул. Размер циклического буфера передачи должен быть выбран так, чтобы при достижении данными пороговой отметки в нем было достаточно места для другого полного дискового блока. Прямое чтение с диска в циклический буфер передачи невозможно, так как данные в нем располагаются как бы по кругу. Таким образом, копирование данных и использование оперативной памяти представляют собой компромиссный вариант.

Еще одним фактором сравнения двух подходов является производительность диска. При использовании больших дисковых блоков диск работает на полной скорости, что часто является главной заботой разработчиков. Чтение маленьких Р-кадров и В-кадров по отдельности неэффективно. Кроме того, возможно хранение больших дисковых блоков на чередующихся наборах дисков (описывалось в главе 5), тогда как подобная операция с отдельными кадрами не имеет смысла.

Вариант с использованием маленьких дисковых блоков (см. рис. 7.17, *а*) иногда называют **постоянным шагом времени**, так как каждый указатель индекса соответствует равному количеству миллисекунд фильма. Вариант с использованием больших дисковых блоков (см. рис. 7.17, *б*), напротив, иногда называют **постоянным шагом данных**, так как блоки данных имеют равную длину.

Еще одно отличие в этих двух подходах к организации файлов состоит в том, что если типы кадров хранятся в индексе (см. рис. 7.17, *а*), становится возможной быстрая перемотка при помощи отображения только I-кадров. Однако, в зависимости от частоты присутствия I-кадров в потоке, скорость перемотки может восприниматься либо как слишком быстрая, либо как слишком медленная. В любом случае, при использовании файловой организации, показанной на рис. 7.17, *б*, такой способ быстрой перемотки невозможен. При последовательном чтении файла для выборки требуемых кадров требуется большой объем операций ввода-вывода.

Второй подход заключается в использовании специального файла, воспроизведение которого с нормальной скоростью создает иллюзию перемотки с удесятеренной скоростью. Этот файл может быть структурирован так же, как и другие файлы, с использованием либо кадрового, либо блочного индекса. При открытии файла система должна быть способна быстро находить файл быстрой перемотки. Когда пользователь нажимает кнопку быстрой перемотки, система должна мгновенно найти и открыть файл быстрой перемотки и прыгнуть в правильное место в этом файле. Ей известен номер текущего кадра, по которому, зная коэффициент ускорения файла быстрой перемотки, система может вычислить номер кадра в этом файле.

При использовании кадрового индекса кадры находятся просто по индексу. При использовании блочного индекса требуется дополнительная информация в каждом элементе для идентификации кадра, содержащегося в блоке. Кроме того, требуется выполнение операции двоичного поиска в блочном индексе. Быстрая перемотка назад работает аналогично быстрой перемотке вперед.

Размещение файлов для «почти видео по заказу»

До сих пор мы рассматривали стратегии размещения файлов для видео по заказу. Для «почти видео по заказу» более эффективной оказывается другая стратегия. Как вы помните, один и тот же фильм передается в виде нескольких потоков. Даже если фильм хранится в виде непрерывного файла, для каждого потока требуется перемещение блока головок. Для устранения почти всех подобных операций поиска цилиндра была разработана специальная стратегия файлового размещения [63]. Использование этой стратегии проиллюстрировано на рис. 7.18 на примере фильма, передаваемого с частотой 30 кадров в секунду, с создаваемым каждые 5 мин новым потоком, как в случае на рис. 7.14. При этих параметрах для 2-часового фильма требуется 24 потока.

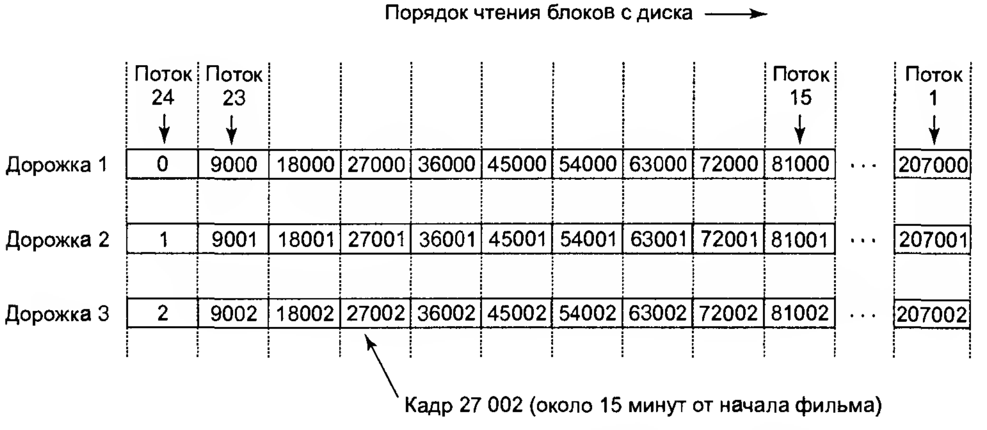


Рис. 7.18. Оптимальное размещение кадров на диске для «почти видео по заказу»

При таком размещении наборы по 24 кадра объединяются и записываются на диск в виде одной записи. Они также могут быть считаны в виде одной записи.

Рассмотрим момент, в который поток 24 только что стартовал. Ему будет нужен кадр 0. Потоку 23, начавшемуся на 5 мин раньше, требуется кадр 9000. Потоку 22 требуется кадр 18 000 и т. д., до потока 0, которому нужен кадр 207 000. Поместив эти кадры на диск именно в таком порядке, видеосервер может удовлетворить требования 24 потоков в обратном порядке всего за одну операцию перемещения блока головок диска. Конечно, кадры могут храниться на диске в обратном порядке, если есть смысл в обслуживании потоков в восходящем порядке. После того как последний поток был обслужен, головка диска может переместиться на дорожку 2, чтобы приготовиться к чтению следующей порции кадров. Такая схема не требует непрерывности всего файла и тем не менее позволяет получить хорошую производительность для нескольких потоков сразу.

Простая стратегия буферизации также заключается в использовании двойного буферирования. В то время пока данные из одного буфера передаются в 24 потока, загружается другой буфер. Когда данные из первого буфера переданы, два буфера меняются местами.

Интересный вопрос состоит в том, насколько большим сделать буфер. Очевидно, он должен вмещать 24 кадра. Однако поскольку у кадров различный размер, верный выбор буфера является нетривиальным делом. Зарезервировать буфер размера, достаточного для вмещения 24 I-кадров, будет чрезмерным, однако если рассчитать размер буфера так, чтобы в него помещалось 24 средних кадра, то будет существовать постоянная опасность, что этого буфера не хватит.

К счастью, для каждого конкретного фильма максимальный размер дорожки (в смысле рис.7.18) можно сосчитать заранее и выбрать буфер точно такого размера. Однако может случиться, что в дорожке максимального размера будет 16 I-кадров, тогда как следующая по размеру дорожка будет содержать всего 9 I-кадров. Возможно, более разумная стратегия заключается в выборе буфера по второй по размеру дорожке. Такая стратегия означает усечение самой большой дорожки при игнорировании одного кадра для некоторых потоков. Чтобы избежать мелькания, можно повторять отображение предыдущего кадра. Никто этого не заметит.

Если продолжить эту идею, еще лучше, возможно, остановиться на размере третьей по величине дорожки, таким образом, используя буфер, способный вместить, скажем, 4 I-кадра и 20 P-кадров. Повтор двух кадров в 2-часовом фильме, возможно, будет вполне приемлемым. До какого предела можно развивать данную идею? Вероятно до того момента, когда буфер сможет вмещать около 99 % всех дорожек с кадрами. В данном случае мы получаем выбор между необходимым объемом оперативной памяти и качеством демонстрации фильма. Следует заметить, что чем больше потоков одновременно передает видеосервер, тем лучше окажутся статистические показатели и тем более однородным окажется набор кадров.

Размещение нескольких файлов на одном диске

До сих пор рассматривался вопрос размещения на диске одного фильма. Конечно, на видеосервере должно храниться одновременно несколько фильмов. При случайном расположении файлов на диске для перемещения блока головок с одного фильма на другой потребуется много времени.

С этой проблемой можно бороться, принимая в расчет популярность фильмов при размещении их на диске. И хотя мало что может быть сказано о популярности отдельных фильмов (кроме как отметить, что на нее, вероятно, влияют имена суперзвезд), кое-что все-таки можно сказать об относительной популярности фильмов вообще.

Разнообразные оценки популярности видеокассет в пунктах проката, библиотечных книг, web-страниц и даже используемых английских слов в рассказах обнаруживают, что популярность достаточно точно следует определенной закономерности. Эта закономерность была обнаружена гарвардским профессором лингвистики Джорджем Ципфом (1902—1950) и называется теперь **законом Ципфа**. Закон утверждает, что если выстроить элементы (видеокассеты, книги, web-страницы и т. д.) по порядку их популярности, то вероятность того, что следующий клиент выберет k-й элемент списка, примерно равна C/k, где C — нормирующая константа.

Таким образом, частоты попаданий в тройку лидеров составляют соответственно $C/1$, $C/2$ и $C/3$, где C вычисляется так, чтобы сумма всех слагаемых была равна 1. Другими словами, при N фильмах получим:

$$C/1 + C/2 + C/3 + C/4 + \ldots + C/N = 1.$$

Из этого уравнения можно определить значение C. Для наборов из 10, 100, 1000 и 10 000 элементов эти значения соответственно равны 0,341, 0,193, 0,134 и 0,102. Например, для 1000 фильмов частоты пяти самых популярных фильмов равны 0,134, 0,067, 0,045, 0,034 и 0,027.

Закон Ципфа проиллюстрирован на рис. 7.19. В данном случае мы решили продемонстрировать его силу на численности населения 20 крупнейших городов США. Согласно закону Ципфа, численность населения второго по величине города должна быть приблизительно в два раза ниже, чем у самого крупного города, в третьем по величине городе должно жить около трети от численности населения самого крупного города[1] и т. д. Хотя и не идеально, но удивительно точно кривая, описывающая закон Ципфа, ложится рядом с точками на графике.

Для фильмов, хранящихся на видеосервере, закон Ципфа утверждает, что самый популярный фильм будут брать в два раза чаще, чем второй по популярности, в три раза чаще, чем третий и т. д. Хотя поначалу кривая распределения частот стремительно убывает, у нее довольно длинный «хвост». Например, популярность 50-го фильма составляет $C/50$, а популярность 51-го фильма — $C/51$, что означает всего 2-процентное отличие в популярности между 50-м и 51-м фильмами. Чем дальше мы будем продвигаться по «хвосту» кривой, тем меньшим будет относительное различие между популярностью соседних фильмов. Из этого можно сделать вывод, что на видеосервере необходимо хранить довольно большое количество фильмов, так как спрос на фильмы за пределами 10 самых популярных достаточно значителен.

Зная относительную популярность различных фильмов, можно смоделировать производительность видеосервера и использовать эту информацию для размеще-

[1] Для справки: Нью-Йорк (1990) 7 322 564; Лос-Анджелес (1994) 3 620 543; Чикаго (1992) 2 832 901 жителей. — *Примеч. перев.*

ния файлов. Изучения показали, что наилучшая стратегия на удивление проста и не зависит от распределения частот. Эта стратегия называется **органным алгоритмом** ([140, 361]). Он заключается в размещении наиболее популярных фильмов в середине диска, второго и третьего в хит-параде фильмов по обеим сторонам от самого популярного. Затем располагаются четвертый и пятый фильмы и т. д., как показано на рис. 7.20. Такое расположение файлов лучше всего работает, если каждый фильм представляет собой непрерывный файл того типа, который показан на рис. 7.16, но также может, хотя и с меньшей эффективностью, применяться и для сегментированных файлов, расположенных в узком интервале соседних цилиндров диска. Название алгоритма происходит от того, что гистограмма частот напоминает слегка скособоченный орган.

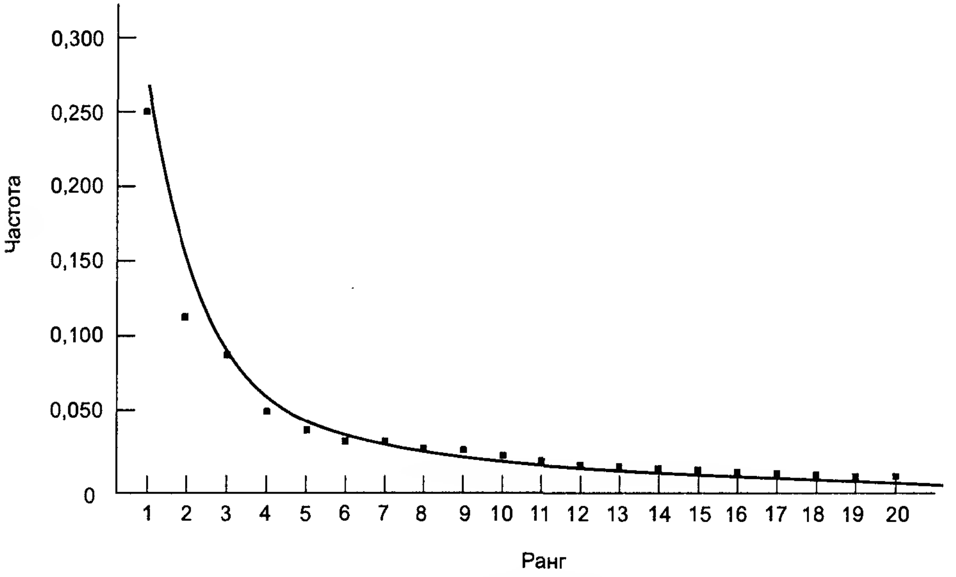


Рис. 7.19. Кривая закона Ципфа для N = 20

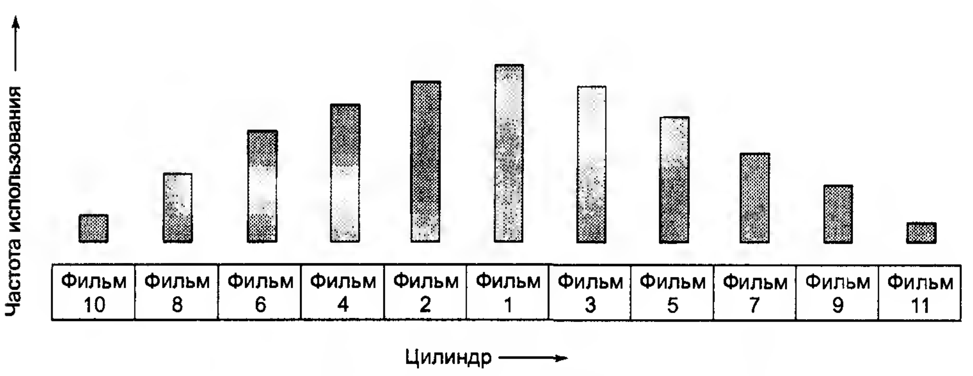


Рис. 7.20. Органное распределение файлов на видеосервере

Таким образом, этот алгоритм пытается удерживать головку диска посреди диска. При 1000 фильмах по закону Ципфа вероятность заказа одного из пяти наиболее популярных фильмов составит 0,307. Это означает, что примерно 30 % времени блок головок будет находиться над цилиндрами, содержащими пятерку самых популярных фильмов, что удивительно много при 1000 фильмах на диске.

Размещение файлов на нескольких дисках

Для достижения более высокой производительности на видеосерверах часто используется несколько дисков, работающих параллельно. Иногда используются чередующиеся наборы RAID, но не часто, так как в общем случае система RAID предоставляет более высокую надежность за счет производительности. Видеосерверам, как правило, требуется высокая производительность, и они обычно не сильно заботятся об исправлении временных ошибок. К тому же контроллеры RAID могут стать узким местом системы, если им придется поддерживать одновременную работу большого числа дисков.

Более распространенная конфигурация представляет собой просто большое количество дисков, иногда называемое **дисковой фермой**. Эти диски не вращаются синхронно и не содержат битов четности, как диски RAID-систем. Одна возможная конфигурация заключается в помещении фильма *A* на диск 1, фильма *B* на диск 2 и т. д., как показано на рис. 7.21, *а*. На практике современные диски могут вмещать по нескольку фильмов.

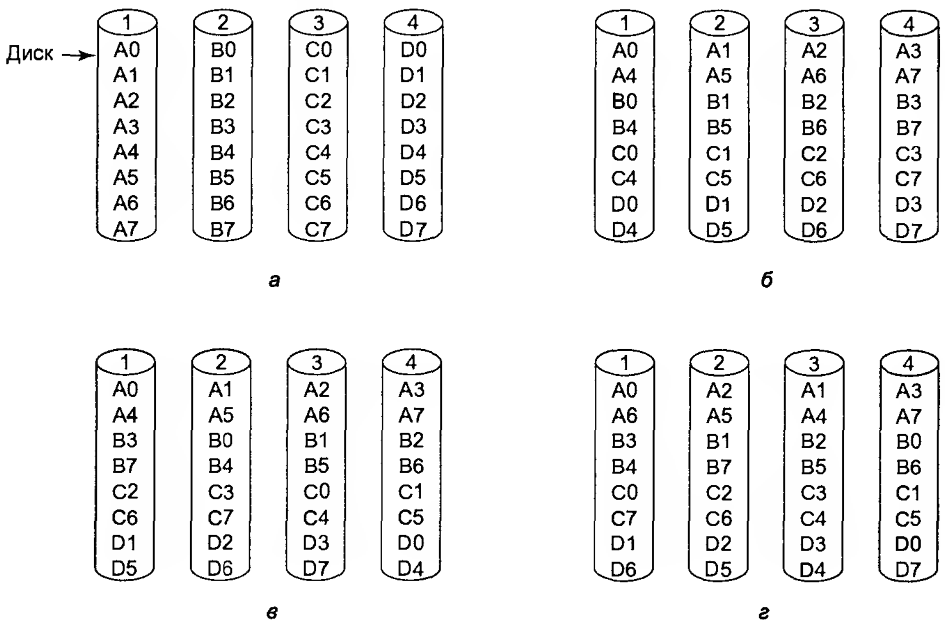


Рис. 7.21. Четыре способа организации мультимедийных файлов на нескольких дисках: без чередования (*а*); с одинаковым чередованием для всех файлов (*б*); в шахматном порядке (*в*); со случайным чередованием (*г*)

Такая организация проста в реализации и обладает не очень высокой надежностью: если один диск выйдет из строя, все фильмы на дисковой ферме станут недоступны. Обратите, однако, внимание, что потеря диска с фильмами не идет ни в какое сравнение с потерей диска с какой-либо базой данных, так как фильмы можно легко снова записать на запасной диск с DVD. Недостаток такого подхода заключается в том, что нагрузка на диски может оказаться неравномерной. Если на одних дисках будут храниться фильмы, пользующиеся большим спросом, а на других дисках — менее популярные фильмы, эффективность системы окажется низкой. Конечно, при известной частоте запросов к фильмам можно переместить некоторые из них на другие диски, чтобы сбалансировать нагрузку.

Второй вариант организации заключается в том, что каждый фильм распределяется по нескольким дискам (рис. 7.21, *б*). Предположим, что все кадры имеют один и тот же размер (то есть несжатые кадры). Фиксированное число байтов фильма *A* пишется на диск 1, затем такое же количество байтов записывается на диск 2 и т. д., до последнего диска (злемент *A*3). Затем элемент *A*4 чередующегося набора снова пишется на диск 1 и т. д., пока не будет записан весь файл. Затем на те же диски таким же образом пишутся фильмы *B*, *C* и *D*.

Возможный недостаток такого подхода — поскольку все фильмы начинаются на первом диске, нагрузка на диски может оказаться несбалансированной. Один из способов более равномерного распределения нагрузки между дисков состоит в перемешивании стартовых дисков в шахматном порядке (рис. 7.21, *в*). Еще одна попытка сбалансировать нагрузку заключается в использовании случайного чередования фильмов по дискам (рис. 7.21, *г*).

До сих пор мы предполагали, что все кадры имеют одинаковый размер. Для фильмов в формате MPEG-2 такое предположение является неверным: I-кадры значительно превосходят P-кадры. С этой проблемой можно бороться двумя способами: покадровым чередованием и поблочным чередованием. При покадровом чередовании первый кадр фильма *A* пишется на диск 1 в виде непрерывного фрагмента, независимо от его размеров. Следующий кадр записывается на диск 2 и т. д. Фильм *B* распределяется по дискам таким же способом, начинаясь либо на том же диске, либо на следующем (при шахматном порядке), либо на случайном диске. Поскольку считываются кадры по одному, такое чередование не ускоряет считывание одного отдельно взятого фильма. Однако при этом нагрузка распределяется по дискам лучше, чем на рис. 7.21, *а*, где эффективность может оказаться довольно низкой, если много зрителей захотят одновременно смотреть фильм *A*, но никто не захочет смотреть фильм *C*. В целом распределение нагрузки по нескольким дискам позволяет лучше использовать общую пропускную способность дисков, тем самым давая возможность увеличить количество обслуживаемых одновременно клиентов.

Другой вариант состоит в поблочном чередовании. Каждый фильм разбивается на блоки фиксированного размера, которые и записываются последовательно (или в случайном порядке) на каждый диск. Каждый блок содержит один или более кадров или фрагментов кадров. В таком случае система может давать запросы на одновременное считывание нескольких блоков одного фильма. В результате каждого запроса данные считываются в разные буферы памяти, но таким образом, что после выполнения всех запросов в оперативной памяти собирается непрерыв-

ный фрагмент фильма (содержащий несколько кадров). Эти запросы могут обрабатываться параллельно. При удовлетворении последнего запроса запрашиваемому процессу может быть подан сигнал о том, что работа завершена. После этого процесс может передать эти данные пользователю. Несколько кадров спустя, когда в буфере останутся несколько последних кадров, издаются следующие запросы на чтение новых данных в другом буфере. При таком подходе под буферы используется большое количество памяти, чтобы поддерживать диски в постоянно работающем состоянии. В системе с 1000 активных пользователей и 1-мегабайтными буферами (например, с использованием 256-килобайтных блоков на каждом из четырех дисков) для буферов требуется 1 Гбайт ОЗУ. Для видеосервера с 1000 пользователями такая величина — просто семечки, и она не должна стать проблемой.

Последний вопрос, касающийся чередования, заключается в том, сколько дисков должно входить в чередующийся набор. Например, при 2-гигабйатных фильмах и 1000 дисках на каждый диск может быть записан блок в 2 Мбайт, таким образом, ни один фильм не будет дважды использовать один и тот же диск. Другая крайность заключается в разбиении дисков на небольшие группы (как на рис. 7.21), когда каждый фильм распределяется по дискам внутри одной группы. Первый вариант, называемый **широким чередованием**, хорошо распределяет нагрузку по дискам. Основной его недостаток состоит в том, что в случае выхода из строя хотя бы одного диска ни один фильм не может демонстрироваться. Недостатком второго варианта, называемого **узким чередованием**, является неравномерность популярности фильмов, зато потеря одного диска разрушит фильмы только в одной небольшой группе дисков. Чередующиеся наборы с кадрами переменных размеров детально анализируются в [300].

Кэширование

Традиционные алгоритмы удаления из кэша наиболее давно использовавшихся элементов, называемые также алгоритмами LRU-кэширования (Least-Recently-Used — с наиболее давним использованием), плохо годятся для работы с мультимедийными файлами, так как характеристики доступа для мультимедийных и текстовых файлов различаются. Идея традиционных алгоритмов LRU-кэширования заключается в том, что после использования блока он хранится в кэше на случай, если этот блок вскоре снова понадобится. Например, при редактировании файла набор блоков, в которых хранится файл, используется снова и снова, пока не будет завершен сеанс редактирования файла. Другими словами, если имеется относительно высокая вероятность повторного использования блока в течение короткого интервала времени, этот алгоритм стоит использовать, чтобы избежать необходимости обращения к диску в будущем.

В мультимедиа обычный рисунок доступа к файлу таков, что фильм просматривается последовательно от начала до конца. Повторно блоки используются редко (только когда пользователь перематывает фильм, чтобы второй раз посмотреть какую-либо сцену). Следовательно, обычная техника кэширования не работает. Тем не менее в мультимедиа кэширование может применяться, но оно должно использоваться по-другому. В следующих разделах мы рассмотрим технику кэширования в мультимедиа.

Блочное кэширование

Хотя простое хранение блока на всякий случай не имеет смысла в мультимедиа, но предсказуемость поведения мультимедийных систем может быть использована для применения кэширования. Предположим, что два пользователя смотрят один и тот же фильм, причем один из них начал просмотр на 2 с позже другого. После того как первый пользователь получил и просмотрел некий блок, весьма вероятно, что спустя 2 с этот блок понадобится другому пользователю. Система легко может отличить фильмы, просматриваемые только одним зрителем, от фильмов, у которых много зрителей, с близко расположенными друг другу точками воспроизведения.

Таким образом, в данной ситуации возникает возможность сохранять в кэше блоки первого зрителя, так как с большой вероятностью они понадобятся второму зрителю. Хранить эти блоки в кэше или не хранить, зависит от времени хранения и наличия свободной памяти. Вместо хранения в кэше всех блоков диска подряд и удаления из кэша наиболее давно использовавшегося блока должна использоваться совершенно другая стратегия. Каждый фильм, у которого имеется второй зритель, расположенный в пределах интервала времени ΔT от первого зрителя, должен помечаться как фильм, для которого возможно кэширование и все его блоки должны сохраняться в кэше до тех пор, пока их не просмотрит второй (и, возможно, третий) пользователь. Для остальных фильмов кэширование не выполняется.

Эту идею можно развить. В некоторых случаях бывает возможно объединить два потока. Предположим, два пользователя смотрят один фильм с интервалом в 10 с. Удерживать блоки в кэше в течение 10 с можно, но для этого требуется много памяти. В качестве альтернативы может использоваться не совсем честный прием: синхронизовать эти два фильма. Этого можно добиться, если слегка изменить частоту кадров обоих фильмов. Идея проиллюстрирована на рис. 7.22.

На рис. 7.22, *а* оба фильма передаются со стандартной для NTSC частотой 1800 кадров в минуту. Поскольку пользователь 2 начал просмотр на 10 с позже, он так и продолжает отставать на эти 10 с весь фильм. Однако на рис. 7.22, *б* при появлении второго зрителя поток первого пользователя слегка притормаживается. Вместо 1800 кадров в минуту в течение следующих 3 мин фильм передается с частотой 1750 кадров в минуту. Через 3 мин этот поток передает кадр 5550. Кроме того, поток 2-го пользователя воспроизводится с частотой, увеличенной до 1850 кадров в минуту, и через 3 мин он также достигает кадра 5550. С этого момента оба потока продолжают передачу с нормальной скоростью.

Во время выравнивания потоков скорость обоих потоков изменяется на 2,8 %. Маловероятно, что пользователи заметят это[1]. Однако, если это так важно, период синхронизации может быть увеличен.

Альтернативный способ снижения скорости пользователя для объединения его потока с другим потоком заключается в том, чтобы предоставить пользователю возможность смотреть фильмы с рекламой, естественно, за меньшую плату, чем без рекламы. Пользователь также сможет выбирать категории рекламируемой про-

[1] 2,8 % это чуть меньше четверти тона музыкального ряда. Если в фильме в момент изменения скорости (в любую сторону) будет играть музыка, что бывает довольно часто, то эффект будет заметен и весьма неприятен. — *Примеч. перев.*

дукции, поэтому реклама будет не столь раздражающей, и вероятность согласия пользователя на ее показ будет выше. Изменяя количество, длительность и время показа рекламных роликов, можно синхронизировать потоки, расхождение которых по времени значительно больше 10 с [187].

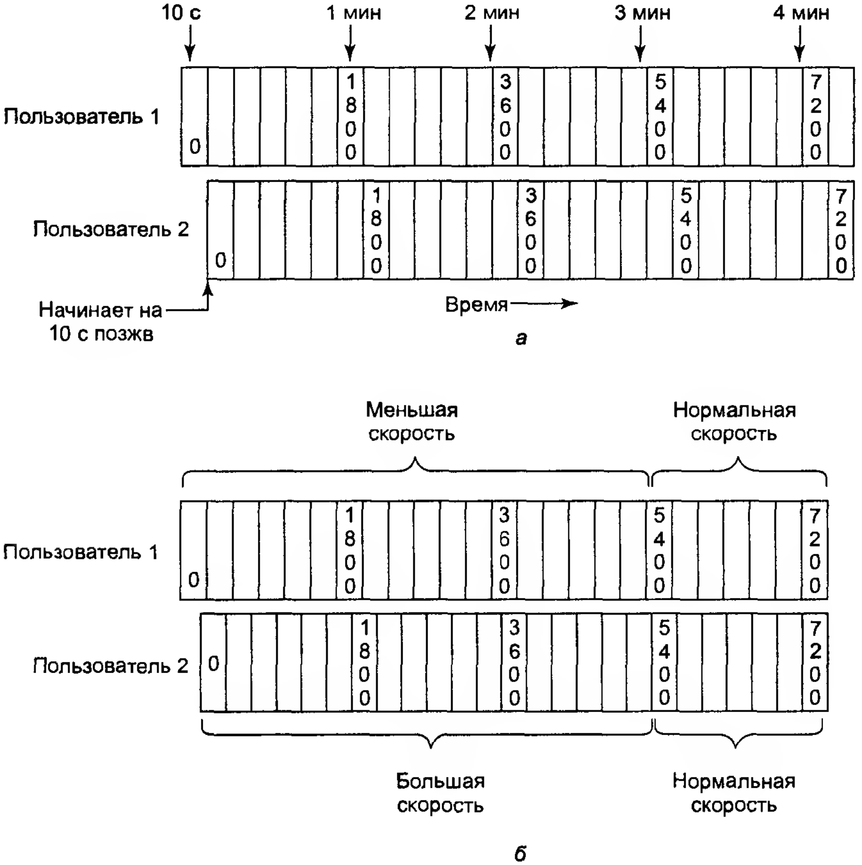


Рис. 7.22. Два пользователя смотрят один и тот же фильм, рассинхронизированный на 10 с (*а*); объединение двух потоков в один (*б*)

Файловое кэширование

Кэширование может применяться в мультимедийных системах и другим образом. Из-за огромного размера большинства фильмов (2 Гбайт) видеосерверы часто не способны хранить все свои фильмы на жестком диске, поэтому они хранят их на DVD или на ленте. Когда требуется фильм, он всегда может быть скопирован на диск, но чтобы найти фильм и скопировать его на диск, требуется значительное время. Соответственно, большинство видеосерверов поддерживают дисковый кэш фильмов, пользующихся наибольшим спросом. Популярные фильмы целиком хранятся на жестких дисках.

Другой способ использования кэша состоит в хранении на жестком диске первых нескольких минут каждого фильма. Таким образом, при запросе фильма воспроизведение может начинаться немедленно с файла на жестком диске. Тем временем фильм копируется на жесткий диск с DVD или ленты. Если каждый раз сохранять на жестком диске достаточный по размеру фрагмент фильма, вероятность того, что программа успеет скопировать следующий фрагмент, будет довольно высока. При этом считанный фрагмент будет попадать в кэш и оставаться на диске на тот случай, если он вдруг понадобится позднее. Если этот фильм окажется невостребованным в течение долгого времени, он будет удален из кэша, чтобы освободить место для более популярных фильмов.

Дисковое планирование в мультимедиа

Мультимедиа накладывает на диски требования, отличные от требований традиционных текст-ориентированных приложений, таких как компиляторы или текстовые процессоры. В частности, мультимедиа требует исключительно высоких скоростей передачи данных и доставки данных в режиме реального времени. Ни одна из этих задач не является тривиальной. Более того, в случае видеосервера присутствует экономическое давление, нацеленное на то, чтобы один сервер одновременно обслуживал тысячи клиентов. Эти требования накладывают отпечаток на всю систему. Выше была рассмотрена файловая система. Теперь познакомимся с планированием в мультимедийных системах.

Статическое дисковое планирование

Мультимедиа предъявляет ко всем частям системы повышенные требования по скорости передачи данных и доставке данных в режиме реального времени. В то же время у мультимедийных систем есть одно свойство, упрощающее управление этими системами. Таким свойством является предсказуемость. В традиционной операционной системе запросы к блокам диска поступают почти непредсказуемым образом. Лучшее, что может сделать дисковая подсистема, — это выполнить опережающее чтение по одному блоку для каждого открытого файла. В остальном ей остается лишь ожидать запросов и выполнять их по мере поступления. В мультимедийных системах все по-другому. Каждый активный поток накладывает на систему предсказуемую, строго определенную нагрузку. При воспроизведении фильма в формате NTSC каждые 33,3 мс каждому клиенту требуется следующий кадр из его файла, и у системы есть 33,3 мс, чтобы считать и переслать всем клиентам по кадру (система должна буферировать по крайней мере по одному кадру на каждый поток, чтобы при считывании кадра $k + 1$ она могла параллельно передавать кадр k).

Эта предсказуемость нагрузки может использоваться для планирования дисковых операций с помощью алгоритмов, специально скроенных для мультимедийных систем. Ниже мы рассмотрим всего один диск, но данный подход применим также и для нескольких дисков. В этом примере мы предположим, что имеется всего 10 пользователей, каждый из которых смотрит свой фильм. Более того, мы будем считать, что у всех фильмов одинаковое разрешение, частота кадров и другие характеристики.

В зависимости от системы на компьютере могут работать 10 процессов, по одному на видеопоток, или один процесс с 10 потоками, или даже один процесс с одним потоком, поочередно обрабатывающий 10 видеопотоков. Эти детали не важны. Важно лишь то, что время делится на **раунды**, то есть интервалы времени, равные длительности воспроизведения одного кадра (33,3 мс для NTSC или 40 мс для PAL). В начале каждого раунда от имени каждого пользователя формируется дисковый запрос (рис. 7.23).

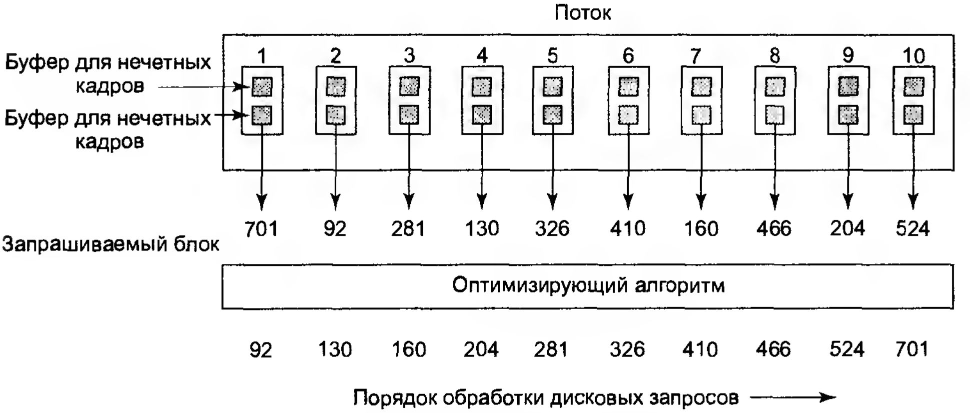


Рис. 7.23. *За один раунд каждый фильм запрашивает один кадр*

Получив к началу раунда все запросы, диск знает, что ему предстоит делать в течение этого раунда. Он также знает, что до начала следующего раунда никаких других запросов не поступит. Соответственно, он может сортировать запросы так, чтобы обрабатывать их оптимальным образом, возможно, в порядке цилиндров (хотя в некоторых случаях может использоваться обработка в порядке секторов). На рис. 7.23 запросы показаны отсортированными в порядке цилиндров.

На первый взгляд может показаться, что подобная оптимизация дисковых операций не нужна, так как до тех пор, пока диск успевает выполнять запросы в срок, не имеет значения, остается ли у него в запасе 1 мс или 10 мс. Однако подобное рассуждение является ошибочным. Оптимизация подобного рода позволяет снизить среднее время обработки каждого запроса, это означает, что диск может обработать большее число запросов за один раунд. Другими словами, подобная оптимизация обработки дисковых запросов позволяет увеличить количество фильмов, одновременно транслируемых видеосервером. Кроме того, оставшееся время в конце раунда также может быть использовано для обслуживания любых запросов, не являющихся запросами реального времени.

Если у сервера слишком много потоков данных, время от времени он не будет успевать считывать кадры из удаленных уголков диска. Но пока такие неудачи будут редкими, с ними можно мириться, выигрывая в количестве одновременно обслуживаемых потоков данных. Обратите внимание, что для работы диска значение имеет число потоков данных, а не число клиентов. Если несколько клиентов принимают один и тот же поток, это никак не влияет на производительность диска или на планирование дисковых операций.

Для поддержания равномерности поступления данных к клиентам на сервере должна использоваться двойная буферизация. Во время первого раунда используется один набор буферов, по буферу на поток данных. Когда раунд завершен, выходной процесс или процессы разблокируются и начинают передавать первый кадр. В то же время поступают новые запросы на кадр 2 каждого фильма (для каждого фильма у процесса может быть по два потока: один для ввода с диска и один для вывода). Такая схема уменьшает количество дисковых операций вдвое за счет удвоения объемов оперативной памяти, занимаемой буферами. В зависимости от относительной доступности, производительности и стоимости операций ввода-вывода в памяти против дискового ввода-вывода может быть рассчитана и использована оптимальная стратегия.

Динамическое дисковое планирование

В приведенном выше примере предполагалось, что у всех потоков данных одинаковое разрешение, частота кадров и другие характеристики. Отбросим теперь данное предположение. Различные фильмы могут обладать различными скоростями передачи данных, поэтому нельзя разбивать время на раунды по 33,3 мс, за которые для каждого потока считывается по кадру. Запросы к диску поступают более или менее случайно.

Каждый дисковый запрос указывает номер блока, который должен быть прочитан, а также крайний срок выполнения зтой операции. Для простоты будем предполагать, что фактическое время обслуживания каждого запроса одинаково (даже если на самом деле это не так). Таким образом, мы можем вычесть фиксированное время обслуживания каждого запроса из его конечного срока, чтобы получить крайний срок инициирования запроса. Такое предположение упрощает модель, так как планировщик дисков беспокоится о выполнении запросов в срок.

При запуске системы дисковых запросов, ждущих обработки, нет. Когда приходит первый дисковый запрос, он обслуживается немедленно. При выполнении первой операции по поиску цилиндра могут поступить другие запросы, поэтому, когда первый запрос выполнен, у дискового драйвера может быть выбор, какой запрос обрабатывать следующим. Один из запросов выбирается и обслуживается. Когда этот запрос выполнен, у нас есть набор необслуженных запросов: те, что не были обслужены в первый раз плюс поступившие во время обработки второго запроса. В общем случае после завершения обработки запроса у драйвера есть некий набор необработанных запросов, из которых он должен выбрать следующий. Вопрос звучит так: «Какой алгоритм следует использовать для выбора следующего запроса?»

В выборе следующего дискового запроса играют роль два фактора: крайние сроки завершения и цилиндры. С точки зрения производительности сортировка запросов по цилиндрам и использование элеваторного алгоритма минимизируют время поиска цилиндра, но при этом запросы к крайним цилиндрам диска могут оказаться обслуженными с опозданием. С точки зрения задачи реального времени следует сортировать запросы по срокам выполнения. При этом минимизируется вероятность пропуска крайнего срока обработки запроса, но увеличивается суммарное время поиска.

Оба эти фактора учитываются алгоритмом **scan-EDF** [273]. Основная идея этого алгоритма заключается в объединении запросов, конечные сроки которых расположены близко на временной шкале, в пакеты и обработка этих пакетов в порядке цилиндров. Рассмотрим ситуацию на рис. 7.24 в момент времени $t = 700$. Дисковый драйвер знает, что у него есть 11 необработанных запросов к различным цилиндрам и с различными сроками выполнения. Он может, например, решить обрабатывать пять запросов с наименьшими сроками выполнения в виде одного пакета, отсортировать их по порядку цилиндров и использовать для их обработки элеваторный алгоритм. При этом порядок считываемых цилиндров будет следующим: 110, 330, 440, 676 и 680. До тех пор пока каждый запрос выполняется в срок, организация запросов может изменяться так, чтобы минимизировать требуемое время перемещения блока головок.



Рис. 7.24. Обработка пакетов при помощи алгоритма scan-EDF

Когда у различных потоков различные скорости потоков данных, появление нового клиента вызывает серьезную проблему. Видеосервер должен принять решение, может ли он себе позволить обслуживание еще одного пользователя. Если допуск нового клиента приведет к тому, что другие потоки данных станут часто пропускать свои критические сроки, то ответом клиенту, вероятно, должен быть отказ. Существует два метода принятия решения, разрешить ли подключение нового клиента. Один способ состоит в предположении, что каждому среднему клиенту требуется определенное количество ресурсов, например пропускная способность диска, буферы памяти, время центрального процессора и т. д. Если всех этих ресурсов достаточно для одного среднего клиента, то новый клиент получает разрешение на просмотр фильма.

Другой алгоритм учитывает большее количество деталей. Он изучает характеристики конкретного фильма, заказываемого клиентом, проверяя (заранее рассчитанные) скорость передачи данных для этого фильма, отличающуюся для черно-белых и цветных фильмов, мультипликационных и кинофильмов и даже для любовных историй и фильмов о войне. Мелодрамы обычно развиваются неторопливо с долгими сценами и медленными наплывами кадров и поэтому отлично сжимаются, тогда как в кинобоевиках много динамичных сцен и быстро сменяющих друг друга монтированных сцен, снятых с разных камер, поэтому в них много I-кадров и большие P-кадры. Если у сервера достаточно мощностей для демонстрации конкретного фильма, заказываемого клиентом, тогда клиенту выдается разрешение на просмотр; в противном случае клиент получает отказ.

Исследования в области мультимедиа

Мультимедиа находится сегодня в центре внимания, поэтому в этой области проводится большое количество исследований. Большая часть этих исследований посвящена содержанию мультимедиа, инструментарию для создания мультимедиа и приложениям. Все эти темы выходят за пределы данной книги. Однако некоторые из исследований касаются структуры операционной системы, либо созданию новой операционной системы [41], либо добавлению мультимедийной поддержки к существующей операционной системе [234]. Близкой темой исследований является разработка мультимедийных серверов [26, 153, 216, 362].

Некоторые статьи по мультимедийной тематике посвящены не новым системам, а алгоритмам, применяемым в мультимедийных системах. Популярной темой является планирование центрального процессора реального времени для мультимедиа [18, 38, 83, 134, 167, 247, 364]. Другой освещаемой в статьях темой является планирование дисковых операций в мультимедийных системах [200, 279, 355]. Размещение файлов и управление нагрузкой на видеосерверах тоже представляет собой важную тему исследований [123, 300, 343], как и объединение видеопотоков для снижения необходимой пропускной способности [103].

В тексте обсуждалось влияние популярности фильмов на размещение файлов на видеосервере. Эта тема является областью продолжающихся исследований [34, 138]. Наконец, безопасность и конфиденциальность в мультимедиа (например, на видеоконференциях) также представляют интерес для исследователей [6, 157].

Резюме

Мультимедиа представляет собой развивающуюся область применения компьютеров. Из-за больших размеров мультимедийных файлов и требований реального времени по их доставке операционные системы, разработанные для работы с текстом, не являются оптимальными для мультимедиа. Мультимедийные файлы состоят из большого количества параллельных дорожек: как правило, одной видеодорожки и по крайней мере одной звуковой дорожки, иногда также дорожки субтитров. Все эти дорожки должны синхронизироваться при воспроизведении.

Звук записывается при помощи периодического измерения уровня мощности, обычно с частотой 44 100 отсчетов в секунду (для качества звука компакт-диска). К звуковому сигналу может быть применена компрессия, позволяющая уменьшить размеры данных примерно в 10 раз. Для сжатия видеоинформации применяется как внутрикадровое сжатие (JPEG), так и межкадровое сжатие (MPEG). Последний алгоритм представляет P-кадры в виде разностей относительно предыдущего кадра. В алгоритме MPEG также используются B-кадры, базирующиеся либо на предыдущем, либо на последующем кадре.

Для мультимедиа необходимо планирование реального времени, ставящее целью выполнение определенных задач в указанные сроки. Широкое распространение получили два алгоритма. Первый из них является алгоритмом планирования RMS. Он представляет собой статический алгоритм с прерываниями, назначающий процессам фиксированные приоритеты, основываясь на их периодах.

Второй (EDF) представляет собой динамический алгоритм, всегда выбирающий задание с ближайшим предельным сроком выполнения. Алгоритм EDF более сложен, но позволяет достичь 100-процентного использования процессорного времени, что недоступно для алгоритма RMS.

В мультимедийных файловых системах используется, как правило, модель «толкания» данных от сервера, а не «вытягивания» их клиентом. Как только поток запущен, биты поступают с диска без дальнейших запросов пользователя. Такой подход, радикально отличающийся от применяемого в традиционных операционных системах, необходим для выполнения требований реального времени.

Для хранения мультимедийных данных могут использоваться как непрерывные, так и сегментированные файлы. В последнем случае непрерывный сегмент файла может иметь переменную длину (один блок представляет собой один кадр) или фиксированную длину (один блок содержит несколько кадров). У обоих подходов есть свои преимущества и недостатки.

Размещение файлов на диске влияет на производительность. При наличии нескольких файлов на одном диске иногда применяется органный алгоритм. Распространены и чередующиеся наборы файлов (широкие и узкие) на нескольких дисках. Для увеличения производительности также широко применяется блочное и файловое кэширование.

Вопросы

1. Чему равна скорость передачи данных для несжатого полноцветного XGA-монитора, воспроизводящего 25 кадров в секунду? Может ли такой поток поступать с UltraWide SCSI-диска?
2. Может ли несжатый черно-белый телевизионный сигнал формата NTSC передаваться по быстрой сети Ethernet? Если да, то сколько каналов одновременно?
3. Горизонтальное разрешение системы HDTV вдвое превышает обычное телевидение (1280 вместо 640 пикселов). Учитывая информацию, приведенную в тексте, насколько большая пропускная способность требуется для телевидения высокой четкости по сравнению со стандартным телевидением?
4. На рис. 7.2 изображены отдельные файлы для быстрой перемотки вперед и назад. Если видеосервер должен также обеспечивать замедленное воспроизведение в прямом направлении, требуется ли для этого отдельный файл? А для воспроизведения в обратном направлении?
5. Компакт-диск может содержать 74 мин музыки или 650 Мбайт данных. Оцените коэффициент сжатия, используемый для музыки.
6. При квантовании звуковой сигнал преобразуется в 16-разрядное число (один знаковый бит и 15 бит для амплитуды). Чему равен максимальный шум квантования в процентах? Представляет ли он большую проблему при записи концертов для флейты, чем для рок-н-рольных концертов, или эта проблема не зависит от музыкального жанра? Поясните свой ответ.

7. Звукозаписывающая студия может создать оригинальную запись, используя 20-битовое квантование. Конечные потребители их продукта получат 16-битовые данные. Предложите метод снижения эффекта шума квантования и обсудите достоинства и недостатки вашей схемы.
8. Оба формата телевизионных сигналов NTSC и PAL используют канал вещания с полосой частот 6 МГц, в то же время стандарт NTSC предусматривает передачу 30 кадров в секунду, тогда как PAL — только 25 кадров в секунду. Как это возможно? Означает ли это, что при использовании одинаковой схемы кодирования цвета стандарт NTSC имел бы лучшее качество, чем PAL? Аргументируйте свой ответ.
9. В дискретном косинусном преобразовании используется блок 8×8, тогда как в алгоритме, применяемом для компенсации движения, используется блок 16×16. Связаны ли с этим различием какие-либо проблемы, и если да, то как они решаются в стандарте MPEG?
10. На рис. 7.9 было показано, как алгоритм MPEG работает при стационарном заднем плане и двигающемся актере. Рассмотрите ситуацию, в которой видеокамера установлена на треножник и снимает панораму, медленно поворачиваясь слева направо с такой скоростью, что никакие два последовательных кадра не являются одинаковыми. Должны ли все кадры быть I-кадрами? Поясните свой ответ.
11. Предположим, что каждый из трех процессов на рис. 7.10 сопровождается процессом, поддерживающим поток аудиоданных, работающим с тем же периодом, что и видеопроцесс, так что буферы аудиоданных могут обновляться в перерывах между видеокадрами. Все три аудиопроцесса идентичны. Сколько процессорного времени доступно для аудиопроцесса на каждом кадре?
12. На компьютере работают два процесса реального времени. Первый процесс работает каждые 25 мс в течение 10 мс. Второй работает каждые 40 мс в течение 15 мс. Может ли алгоритм RMS управлять этими потоками?
13. Загруженность центрального процессора видеосервера составляет 65 %. Сколько фильмов может демонстрировать такой видеосервер с помощью алгоритма планирования RMS?
14. На рис. 7.12 алгоритм EDF загружает центральный процессор на 100 % вплоть до момента времени $t = 150$. Он не может обеспечить 100-процентную загрузку центрального процессора бесконечно долго, так как в среднем в каждую секунду у алгоритма есть работы только на 975 мс. Продолжите нарисованный график за пределы 150 мс и определите, когда центральный процессор, наконец, в первый раз перейдет в состояние простоя.
15. DVD может хранить достаточно данных для размещения на нем полнометражного фильма со скоростью передачи данных, достаточной для демонстрации фильма качества телевидения. Почему бы не использовать просто «ферму» из нескольких DVD-дисководов в качестве источника данных видеосервера?

16. Оператор системы «почти видео по заказу» обнаружил, что в определенном городе зрители не хотят ждать начала фильма более 6 мин. Сколько параллельных потоков потребуется для 3-часового фильма?
17. Рассмотрите систему, использующую схему Абрам–Профета и Шина, в которой оператор видеосервера хочет предоставить клиентам возможность полностью локального поиска вперед и назад в течение 1 мин. Если предположить, что используется видеопоток MPEG-2 со скоростью передачи данных 4 Мбит/с, сколько понадобится каждому клиенту памяти на локальные буферы?
18. Система видео по заказу HDTV использует модель с маленькими блоками (см. рис. 7.17, *а*), с размером дисковых блоков 1 Кбайт. При разрешении видеоизображения 1280 × 720 и скорости передачи данных 12 Мбит/с сколько дискового пространства будет расходоваться на внутреннюю фрагментацию в 2-часовом фильме, если используется стандарт NTSC?
19. Рассмотрите схему хранения файлов, показанную на рис. 7.17, *а*, для стандартов NTSC и PAL. Для заданных размеров блока и фильма есть ли разница в потерях на внутреннюю фрагментацию для этих двух стандартов? Если да, то какой стандарт лучше и почему?
20. Рассмотрите альтернативы, показанные на рис. 7.17. Обладает ли одна из двух систем преимуществом при переходе на HDTV? Обсудите.
21. Схема видео по заказу Чена и Тапара лучше всего работает в том случае, когда все кадры имеют одинаковый размер. Предположим, что фильм одновременно демонстрируется по 24 потокам и что каждый 10-й кадр является I-кадром. Также предположим, что I-кадры в 10 раз больше P-кадров. B-кадры имеют тот же размер, что и P-кадры. Чему равна вероятность того, что буфера, равного размеру 4 I-кадров и 20 P-кадров, окажется недостаточно? Предположите, что кадры разных типов распределены в потоке случайным образом, независимо друг от друга.
22. Конечный результат примера на рис. 7.15 заключается в том, что точка воспроизведения не находится более в середине буфера. Разработайте схему, гарантирующую 5 мин перед точкой воспроизведения и 5 мин после нее. Вы можете делать любые разумные предположения, но должны объявлять о них открыто.
23. Схема на рис. 7.16 требует, чтобы все языковые дорожки читались при каждом кадре. Допустим, что разработчики видеосервера должны обеспечить поддержку большого числа языков, но не хотят выделять на них много оперативной памяти для буферов. Какие еще альтернативные методы возможны в данной ситуации. Каковы достоинства и недостатки каждого метода?
24. На маленьком видеосервере восемь фильмов. Чему равны вероятности заказа самого популярного фильма, второго по популярности фильма и т. д. в соответствии с законом Ципфа?
25. Для хранения 1000 30-секундных в формате MPEG-2 видеоклипов, демонстрируемых на скорости 4 Мбит/с, используется 14-гигабайтный диск с 1000 цилиндрами. Какую часть времени будет проводить блок головок в 10 средних цилиндрах, учитывая закон Ципфа?

26. Если предположить, что относительный спрос на фильмы *A*, *B*, *C* и *D* описывается законом Ципфа, то чему будет равно относительное использование четырех дисков (см. рис. 7.21) для показанных четырех методов чередования?
27. Два клиента службы видео по заказу начали просмотр фильма в формате PAL с интервалом 6 с. Если система для объединения этих двух потоков в один ускорит один поток и замедлит другой, какой процент ускорения/замедления потребуется, чтобы объединить потоки через 3 мин?
28. На видеосервере с фильмами в формате MPEG-2 используется схема раундов (см. рис. 7.23) для видео в стандарте NTSC. Все видеоданные считываются с одного UltraWide SCSI-диска со средним временем поиска, равным 3 мс. Сколько одновременных потоков может поддерживать такой видеосервер?
29. Повторите предыдущее задание, но с предположением, что алгоритм scan-EDF уменьшает среднее время поиска на 20 %. Сколько одновременных потоков может поддерживать видеосервер теперь?
30. Еще раз повторите предыдущее задание, но на этот раз предположите, что каждый кадр распределен по четырем дискам с алгоритмом scan-EDF, уменьшаемым среднее время поиска на 20 % для каждого диска. Сколько одновременных потоков может поддерживать видеосервер теперь?
31. В тексте описывалось использование пакета из пяти дисковых запросов при планировании ситуации на рис. 7.24. Если обработка всех запросов занимает одинаковое количество времени, чему равно максимальное возможное время выполнения запроса в данном примере?
32. Во многих растровых изображениях, применяемых для создания компьютерных «обоев», используется небольшое количество цветов, и такие изображения легко сжимаются. Простая схема компрессии состоит в следующем: выбирается значение данных, не используемое во входном файле, и используется в качестве флага. Файл считывается побайтно, при этом ищутся повторяющиеся байты. Байты, встречающиеся в файле от одного до трех раз подряд, копируются в выходной файл без преобразований. Повторяющиеся по четыре и более раз байты заменяются флаговым байтом, за которым пишется счетчик байтов от 4 до 255 и сам байт данных. Напишите программу, использующую данный метод сжатия, а также программу декомпрессии, позволяющую восстановить исходный файл. Дополнительный балл: как вы поступите с файлами, содержащими флаговый байт среди данных?
33. Компьютерная анимация реализуется с помощью отображения последовательности слегка отличающихся друг от друга изображений. Напишите программу, рассчитывающую побайтную разницу между двумя несжатыми растровыми изображениями одинакового размера. Выходной файл будет такого же размера, как и входные файлы. Используйте полученный выходной файл в качестве входного для программы сжатия из предыдущего задания и сравните эффективность этого подхода с непосредственным сжатием отдельных изображений.

Глава 8
Многопроцессорные системы

С самого начала компьютерная промышленность была движима нескончаемым стремлением ко все большей и большей вычислительной мощности. Компьютер ENIAC[1] (Electronic Numerical Integrator and Calculator — электронный цифровой интегратор и калькулятор) мог выполнять 300 операций в секунду, что в 1000 раз превосходило скорость всех электромеханических калькуляторов того времени. Однако его колоссальной по тем временам мощности разработчикам показалось недостаточно. Теперь у нас есть машины в миллион раз мощнее, чем ENIAC, однако спрос на все большие мощности не стихает. Астрономы пытаются понять вселенную, биологи бьются над расшифровкой генома человека, а инженеры хотят создать более надежные и экономичные самолеты. Всем им нужны компьютерные мощности. Сколько бы ни было доступно мощности на данный момент, ее всегда оказывается недостаточно.

В прошлом решение всегда состояло в том, чтобы увеличить тактовую частоту процессора. К сожалению, сегодня мы приближаемся к некоторым фундаментальным пределам тактовой частоты. В соответствии со специальной теорией относительности Эйнштейна, никакой сигнал (в том числе электрический) не может распространяться быстрее скорости света, равной 30 см/нс в вакууме и около 20 см/нс в медном проводе или оптоволоконном кабеле. Это означает, что в компьютере с тактовой частотой 10 Гц сигналы за один такт не могут распространяться дальше, чем на 2 см. Для 100-гигагерцового компьютера полная длина пути сигнала будет составлять максимум 2 мм. Компьютер с тактовой частотой 1 ТГц (1000 ГГц) должен иметь размеры менее 100 мкм, чтобы сигнал от одного его конца до другого успел пройти за один такт процессора.

Производство компьютеров такого размера может теоретически и возможно, но при этом мы сталкиваемся с другой фундаментальной проблемой: рассеянием тепла. Чем быстрее работает компьютер, тем больше тепловой энергии он производит, а чем меньше компьютер, тем труднее отводить эту тепловую энергию. Уже сейчас на новых системах с процессором Pentium вентилятор для охлаждения центрального процессора больше, чем сам центральный процессор. Для перехода с частоты 1 МГц на частоту 1 ГГц требовалось всего лишь постепенное усовершенствование

[1] Построен в Великобритании в 1943 году. Долгое время считался первым компьютером в мире. Состоял из 18 000 электронных ламп и был 24 м в длину. — *Примеч. перев.*

технологии производства микросхем. Для перехода на частоту 1 ТГц потребуются более радикальные изменения.

Другой способ увеличения вычислительной мощности системы состоит в использовании параллельных вычислений. Для этого могут использоваться многопроцессорные компьютеры, состоящие из большого количества процессоров, каждый из которых работает с «нормальной» частотой (чтобы это ни означало в данном году). Однако совместно эти процессоры обладают значительно большей мощностью, чем отдельный центральный процессор. В настоящее время серийно выпускаются и продаются системы с 1000 процессорами. В ближайшее десятилетие, вероятно, будет построена система с 1 млн процессоров. Хотя существуют и другие возможные методы увеличения скорости вычислений, например биологические компьютеры, в данной главе мы сконцентрируем наше внимание на системах, состоящих из большого числа обычных центральных процессоров.

Высокопараллельные компьютеры часто применяются для сложных вычислений. Такие задачи, как прогноз погоды, моделирование воздушного потока вокруг крыла самолета, мировой экономики или взаимодействия лекарства с рецептором в мозгу, требуют больших компьютерных мощностей. Для решения этих задач требуются долгие расчеты на большом количестве центральных процессоров одновременно. Многопроцессорные системы, обсуждаемые в данной главе, широко применяются для данных и подобных задач в науке и машиностроении, а также в других областях.

Еще один способ увеличения вычислительной мощности системы заключается в использовании параллельных расчетов на большом количестве отдельных компьютеров, соединенных в локальную или глобальную сеть. В настоящее время продолжается быстрый рост глобальной сети Интернет. Изначально эта сеть создавалась как прототип помехоустойчивой военной системы управления, затем она стала популярной среди ученых кибернетиков, а в последние годы Интернетом начали пользоваться самые широкие слои населения. Не так давно Интернет получил еще одно применение: поскольку он связывает по всему миру тысячи компьютеров, было решено использовать эту сеть для решения больших научных проблем. С точки зрения вычислительной мощности система, состоящая из 1000 компьютеров по всему миру, не отличается от такой же системы из 1000 компьютеров, стоящих в одном помещении, хотя характеризуется задержкой и имеет некоторые другие технические характеристики. Такие системы также будут рассматриваться в данной главе.

Поместить 1 млн не связанных друг с другом компьютеров в одну комнату достаточно легко при условии, что у вас достаточно денег и достаточно большая комната. Разместить 1 млн не связанных друг с другом компьютеров по всему миру еще легче, поскольку не нужно искать большого помещения. Проблемы начинаются, когда вам требуется соединить эти компьютеры друг с другом для решения одной общей задачи. Поэтому была проведена большая работа в области технологии межкомпьютерных соединений, а различные технологии привели к появлению качественно отличных типов систем и различной организации программного обеспечения.

Весь обмен информацией между электронными (или оптическими) компонентами сводится, в конечном итоге, к отправке и приему сообщений, представляющих собой строго определенные последовательности битов. Различия состоят во

временных параметрах, пространственных масштабах и логической организации. Одну крайность составляют мультипроцессорные системы с общей оперативной памятью и с числом процессоров от двух до тысячи. В этой модели каждый центральный процессор обладает равным доступом ко всей физической памяти и может читать и писать отдельные слова с помощью команд LOAD и STORE. Время доступа к памяти обычно составляет от 10 до 50 нс. Хотя такая система, показанная на рис. 8.1, *а*, может показаться простой, ее реализация представляет собой далеко не простую задачу и обычно включает, как мы скоро увидим, большое количество скрытно передаваемых сообщений.

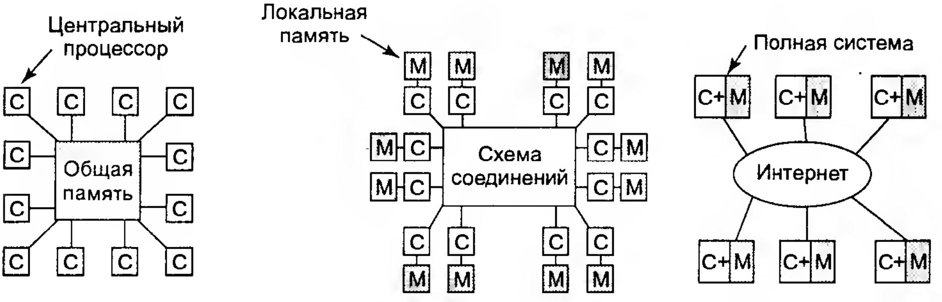


Рис. 8.1. Мультипроцессорная система с общей памятью (*а*); мультипроцессорная система с передачей сообщений (*б*); глобальная распределенная система (*а*)

Следом идут системы (рис. 8.1, *б*), в которых пары, состоящие из центрального процессора и памяти, соединены высокоскоростной соединительной схемой. Такая разновидность системы называется мультипроцессорной системой с передачей сообщений. Каждый блок памяти является локальным для одного центрального процессора, и доступ к ней может получить только этот центральный процессор. Процессоры общаются, обмениваясь сообщениями по соединительной схеме. При хорошем соединении для передачи сообщения может потребоваться от 10 до 50 мкс, что все же значительно больше, чем время доступа к памяти в схеме на рис. 8.1, *а*. В этой схеме нет общей глобальной памяти. Мультикомпьютеры (то есть системы с передачей сообщений) гораздо легче создать, чем мультипроцессоры (системы с общей памятью), но писать программы для них значительно труднее. Поэтому у каждого жанра есть свои поклонники.

Третья модель, показанная на рис. 8.1, *в*, представляет большое количество полноценных компьютеров, соединенных глобальной сетью, такой как Интернет, и образующих вместе **распределенную систему**. У каждого компьютера есть своя собственная память. Компьютеры в распределенной системе общаются, обмениваясь друг с другом сообщениями. Основное различие схем на рис. 8.1, *б* и рис. 8.1, *в* заключается в том, что во втором случае используются полноценные компьютеры, а время передачи сообщений составляет от 10 до 50 мс, то есть примерно еще в 1000 раз больше. Из-за большей величины задержки применение этих **слабосвязанных систем** отличается от использования **сильносвязанных систем**, показанных на рис. 8.1, *б*. Величина задержки у всех трех типов систем отличается друг от друга на три десятичных порядка. Это отличие примерно соответствует разнице между одним днем и тремя годами.

Данная глава состоит из трех основных разделов, соответствующих трем моделям на рис. 8.1. Каждый из этих разделов будет начинаться с краткого вступления, посвященного соответствующему аппаратному обеспечению. Затем будет описываться программное обеспечение, в основном вопросы операционной системы для данного типа систем. Как мы увидим, у каждой системы есть свой круг вопросов.

Мультипроцессоры

Мультипроцессор с общей памятью (или просто мультипроцессор) представляет собой компьютерную систему, в которой два или более центральных процессора делят полный доступ к общей оперативной памяти. Программа, работающая на любом центральном процессоре, видит нормальное (обычно разбитое на страницы) виртуальное адресное пространство. Единственное необычное свойство такой системы заключается в том, что центральный процессор может записать какое-либо значение в память, а затем, считав это слово снова, получить другое значение (потому что другой центральный процессор изменил его). При правильной организации это свойство формирует основу межпроцессорного обмена информацией: один центральный процессор пишет данные в память, а другой считывает их оттуда.

По большей части мультипроцессорные операционные системы представляют собой просто обычные операционные системы. Они обрабатывают системные вызовы, управляют памятью, предоставляют службы файловой системы и управляют устройствами ввода-вывода. Тем не менее есть области, в которых они обладают уникальными свойствами. К этим областям относятся синхрониэация процессов, управление ресурсами и планирование. Ниже сначала будет кратко рассмотрено мультипроцессорное аппаратное обеспечение, а затем мы перейдем к одному из этих вопросов операционной системы.

Мультипроцессорное аппаратное обеспечение

У всех мультипроцессоров каждый центральный процессор может адресоваться ко всей памяти. Однако по характеру доступа к памяти эти машины делятся на два класса. Мультипроцессоры, у которых каждое слово данных может быть считано с одинаковой скоростью, называются **UMA**-мультипроцессорами (Uniform Memory Access — однородный доступ к памяти). В противоположность им мультипроцессоры **NUMA** (NonUniform Memory Access — неоднородный доступ к памяти) этим свойством не обладают. Почему существует такое различие, станет ясно позднее. Сначала будут описаны мультипроцессоры UMA, а затем — мультипроцессоры NUMA.

Архитектура симметричных мультипроцессоров UMA с общей шиной

В основе простейшей архитектуры мультипроцессоров лежит идея общей шины (рис. 8.2, *а*). Несколько центральных процессоров и несколько модулей памяти одновременно используют одну и ту же шину для общения друг с другом. Когда центральный процессор хочет прочитать слово в памяти, он сначала проверяет, свободна ли шина. Если шина свободна, центральный процессор выставляет на нее

адрес нужного ему слова, подает несколько управляющих сигналов и ждет, пока память не выставит нужное слово на шину данных.

Если шина занята, центральный процессор просто ждет, пока она не освободится. В этом заключается проблема данной архитектуры. При двух или трех центральных процессорах состязанием за шину можно управлять. При 32 или 64 центральных процессорах шина будет постоянно занята, а производительность системы будет полностью ограничена пропускной способностью шины. При этом большую часть времени центральные процессоры будут простаивать.

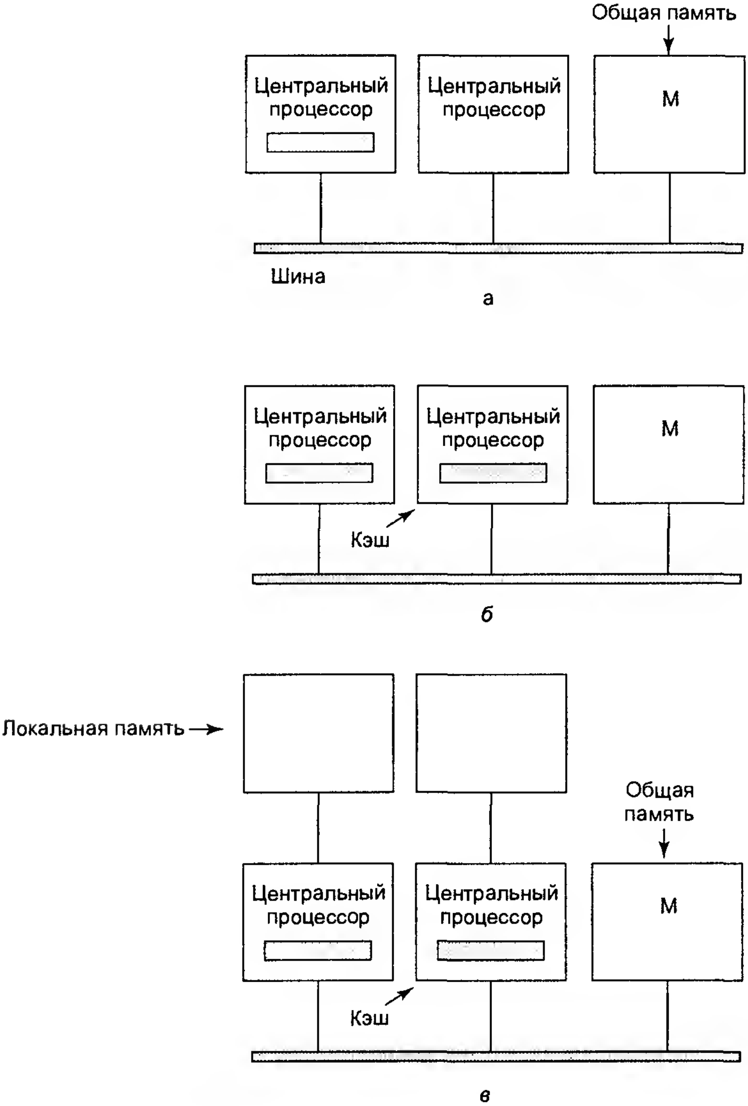


Рис. 8.2. Три варианта архитектуры мультипроцессоров с общей шиной: без кэша (*а*); с кэшем (*б*); с кэшем и собственной памятью (*в*)

Решение этой проблемы состоит в том, чтобы добавить каждому центральному процессору кэш, как показано на рис. 8.2, *б*. Кэш может располагаться внутри микросхемы центрального процессора или рядом с центральным процессором, на процессорной плате. Поскольку большое количество обращений к памяти теперь может быть удовлетворено прямо из кэша, обращений к шине будет существенно меньше, и система сможет поддерживать большее число центральных процессоров. Как правило, кэширование выполняется не для отдельных слов, а для блоков по 32 или по 64 байта. При обращении к слову весь блок считывается в кэш центрального процессора, обратившегося к слову.

Для каждого блока кэша устанавливается режим доступа: либо для него разрешается только чтение (в этом случае этот блок может одновременно присутствовать в нескольких кэшах), либо разрешается и чтение, и запись (в этом случае этот блок не может одновременно присутствовать в нескольких кэшах). При попытке записи центральным процессором слова, находящегося в одном или нескольких удаленных кэшах, аппаратура шины выставляет на шину специальный сигнал, информирующий остальные кэши о записи. Если в остальных кэшах соответствующие блоки «чистые», то есть немодифицированные точные копии блока, находящегося в памяти, тогда они могут просто отбросить свои копии и позволить пишущему процессору получить этот блок из памяти. Если же в каком-либо кэше содержится «грязная» (то есть модифицированная) копия, она должна быть либо записана в память, прежде чем операция записи может быть продолжена, либо передана напрямую пишущему процессору по шине. Существует много протоколов обмена данных между кэшами и памятью.

Еще один вариант архитектуры мультипроцессоров представлен на рис. 8.2, *в*. В этом случае у каждого центрального процессора имеется не только кэш, но также и локальная собственная память, с которой он соединен по выделенной (индивидуальной) шине. Для оптимального использования подобной конфигурации компилятор должен поместить текст программы, константы, стеки (то есть все неизменяемые данные), а также локальные переменные в локальные модули памяти. При этом общая память используется только для общих модифицируемых переменных. В большинстве случаев такая схема использования памяти сильно снижает трафик по шине, но для ее реализации требуются специальные действия со стороны компилятора.

Мультипроцессоры UMA, использующие координатные коммутаторы

Даже при оптимальном использовании кэша наличие всего одной общей шины ограничивает число UMA-мультипроцессоров тридцатью двумя или шестьюдесятью четырьмя. Чтобы преодолеть это ограничение, требуется другая схема соединительной сети. Довольно простая схема соединения n центральных процессоров и k модулей памяти представляет собой **координатный коммутатор** (рис. 8.3). Координатные коммутаторы десятилетиями использовались в телефонной сети для соединения группы входных и группы выходных линий произвольным способом.

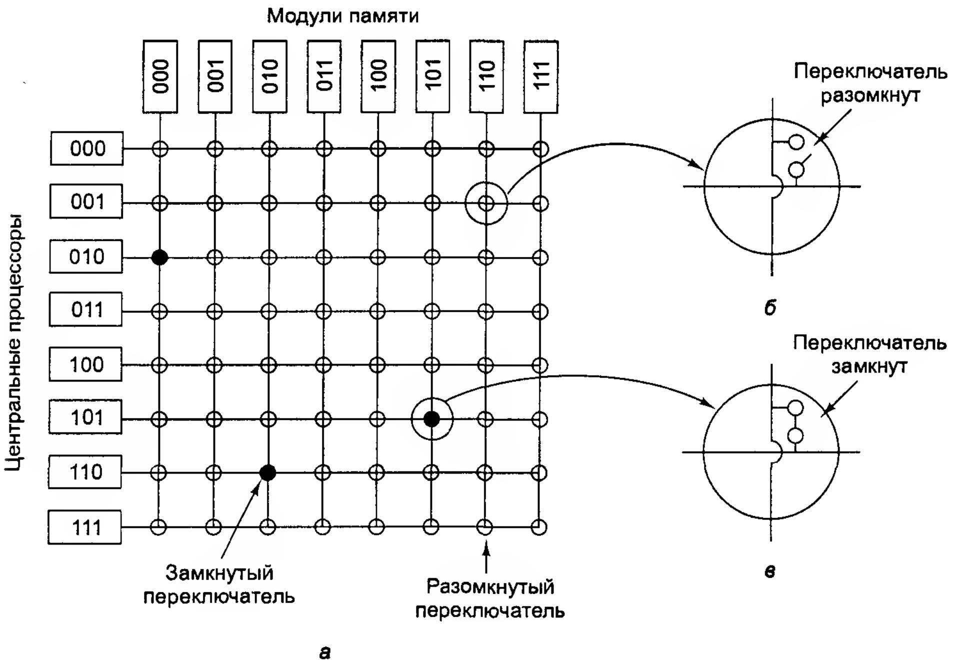


Рис. 8.3. Координатный коммутатор 8 × 8 (*а*); разомкнутый переключатель (*б*); замкнутый переключатель (*в*)

На каждом пересечении горизонтальной (входной) и вертикальной (выходной) линий располагается **координатный переключатель**. Он представляет собой небольшой переключатель, который может быть открыт или закрыт, в зависимости от того, должны быть соединены вертикальная и горизонтальная линии или нет. На рис. 8.3, *а* изображены три одновременно замкнутых переключателя, что позволяет одновременно соединить пары (центральный процессор, блок памяти) (001, 000), (101, 101) и (110, 010). Возможны и другие разнообразные комбинации. Число комбинаций в данном примере равно числу вариантов расположения на шахматной доске восьми ладей таким образом, чтобы они не находились под ударом друг друга.

Одно из самых замечательных свойств координатного коммутатора заключается в том, что он представляет собой **неблокирующую сеть**. Это означает, что ни один центральный процессор не получает отказа соединения по причине занятости какого-либо переключателя (при условии, что сам требующийся модуль памяти свободен). Более того, при такой схеме не требуется планирования доступа к памяти. Даже если семь любых соединений уже установлены, всегда можно соединить оставшийся центральный процессор с оставшимся модулем памяти.

Основной недостаток координатного коммутатора состоит в том, что число переключателей растет пропорционально квадрату от числа центральных процессоров. При 1000 центральных процессоров и 1000 модулях памяти потребуется миллион переключателей. Такой огромный координатный коммутатор просто не реализуем. Тем не менее для систем среднего размера архитектура координатного коммутатора является применимой.

Мультипроцессоры UMA, использующие многоступенчатые коммутаторные сети

Принципиально другая архитектура мультипроцессоров базируется на простых коммутаторах 2×2 (рис. 8.4, *а*). У такого коммутатора два входа и два выхода. Сообщения, поступающие по любой из входных линий, могут переключаться на любую выходную линию. Сообщения в рассматриваемом нами мультипроцессоре будут состоять из четырех частей (рис. 8.4, *б*). Поле *Module* (модуль) указывает модуль памяти. Поле *Address* (адрес) указывает адрес внутри модуля. Поле *Opcode* (код операции) указывает операцию, то есть READ (чтение) или WRITE (запись). Наконец, необязательное поле *Value* (значение) может содержать операнд, например 32-разрядное слово, которое должно быть записано операцией WRITE. По значению поля *Module* коммутатор определяет, по какой из двух выходных линий следует отправить сообщение.

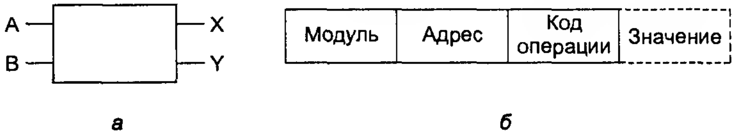


Рис. 8.4. Коммутатор 2×2 (*а*); формат сообщения (*б*)

С помощью данного коммутатора 2×2 можно построить самые различные **многоступенчатые коммутаторные сети** [5, 31, 189]. Один из вариантов таких сетей представляет собой **сеть омега**, показанная на рис. 8.5. Это сеть без излишеств, так сказать, экономический класс сетей. В данном примере она соединяет восемь центральных процессоров с восемью модулями памяти всего 12 коммутаторами. В более общем случае для n центральных процессоров и n модулей памяти потребуется $\log_2 n$ ступеней с $n/2$ коммутаторами в каждой ступени. Таким образом, в целом для сети омега потребуется $(n/2)\log_2 n$ коммутаторов, что значительно меньше, чем n^2 коммутаторов, особенно для больших n.

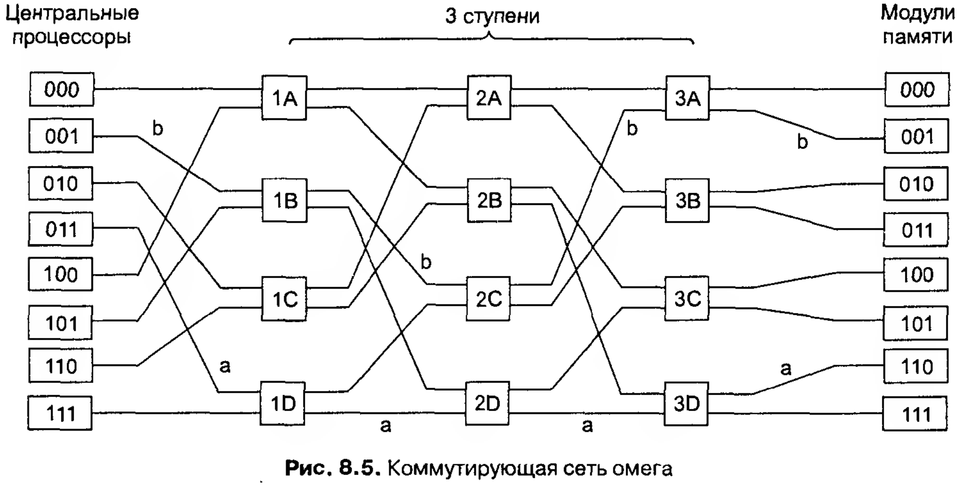


Рис. 8.5. Коммутирующая сеть омега

Схему соединений в сети омега часто называют **идеальным тасованием**, поскольку на каждой ступени перемешивание сигналов напоминает тасование колоды карт. Чтобы понять принципы работы сети омега, предположим, что центральному процессору 011 нужно прочитать слово из модуля памяти 110. Центральный процессор посылает коммутатору 1D сообщение READ, содержащее 110 в поле *Module*. Старший (то есть самый левый) бит этого поля коммутатор использует для выбора маршрута, 0 означает выбор верхнего выхода, а 1 — нижнего. Поскольку бит равен 1, сообщение направляется по нижнему выходу коммутатору 2D.

Все коммутаторы второй ступени, включая коммутатор 2D, используют для маршрутизации второй бит. Он также равен 1, поэтому сообщение передается по нижнему выходу коммутатору 3D. Он проверяет младший бит, и поскольку бит равен 0, то сообщение передается по верхнему выходу и попадает, как и требовалось, к модулю памяти 110. Путь этого сообщения помечен символом *а*.

По мере продвижения по коммутирующей сети левые биты номера модуля оказываются более не нужными. Они могут использоваться для запоминания входных линий, чтобы ответ мог найти обратный путь. Для пути *а* входные линии имеют номера 0 (верхний вход 1D), 1 (нижний вход 2D) и 1 (нижний вход 3D). Ответ направляется по адресу 011, обработка которого производится справа налево.

В то же самое время центральный процессор 001 хочет записать слово в модуль памяти 001. Этот процесс происходит аналогично описанному выше. Сообщение направляется по верхнему, верхнему и нижнему выходу (путь отмечен символом *б*). Когда сообщение доходит до модуля памяти, поле *Module* содержит 001, то есть путь, пройденный сообщением. Поскольку эти два запроса не используют общих коммутаторов, линий и модулей памяти, они могут выполняться параллельно.

Теперь посмотрим, что произойдет, если центральный процессор 000 одновременно с этим захочет обратиться к модулю памяти 000. Его запрос войдет в конфликт с запросом центрального процессора 001 на коммутаторе 3A. Одному из них придется подождать. В отличие от координатного коммутатора, сеть омега представляет собой **блокирующую сеть**. Не все наборы запросов могут быть обработаны одновременно. Возникают конфликты из-за использования линии или коммутатора как между запросами к памяти, так и между ответами памяти на эти запросы.

Было бы желательно распределить запросы к памяти более равномерно между модулями. Один из распространенных методов заключается в использовании младших разрядов в качестве номеров модулей. Представьте, например, байт-ориентированное адресное пространство компьютера, обращающегося к памяти, в основном с 32-разрядными словами. Два младших разряда при этом обычно будут равны 00, но следующие три бита будут распределены равномерно. Если использовать эти три бита в качестве номера модуля, последовательно адресуемые слова окажутся в последовательных модулях. Система памяти, в которой соседние слова хранятся в различных модулях памяти, называется **чередующейся**. Чередующаяся память позволяет добиться максимального распараллеливания, так как большинство обращений к памяти представляют собой запросы по идущим подряд адресам. Возможно создание неблокирующих коммутирующих сетей, предоставляющих каждому центральному процессору несколько путей к каждому модулю памяти для лучшего распределения трафика.

Мультипроцессоры NUMA

Для мультипроцессоров UMA с единственной общей шиной пределом является несколько десятков центральных процессоров, в то время как мультипроцессорам с координатным коммутатором или коммутирующей сетью требуется большое количество (дорогого) аппаратного обеспечения, и количество центральных процессоров в них не намного больше. Чтобы создать мультипроцессор с числом центральных процессоров, превосходящем 100, нужно чем-то пожертвовать. Обычно в жертву приносится идея одинакового времени доступа ко всем модулям памяти. Таким образом, мы получаем концепцию мультипроцессоров NUMA (NonUniform Memory Access — неоднородный доступ к памяти), о которых упоминалось выше. Как и их родственники UMA, мультипроцессоры NUMA предоставляют единое адресное пространство для всех центральных процессоров, но в отличие от UMA-машин доступ к локальной памяти в них быстрее, чем к удаленным модулям. Таким образом, все программы, написанные для UMA, будут работать и на мультипроцессорах NUMA, но их производительность будет ниже, чем на машинах UMA при той же тактовой частоте процессоров.

У машин NUMA есть три ключевые характеристики, которые, взятые вместе, отличают их от других мультипроцессоров.

1. Для всех центральных процессоров имеется единое адресное пространство.
2. Доступ к удаленным модулям памяти осуществляется при помощи специальных команд процессора LOAD и STORE.
3. Доступ к удаленным модулям памяти медленнее, чем к локальной памяти.

В том случае, если доступ к удаленной памяти не является скрытым (то есть кэширование не применяется), система называется **NC-NUMA** (No Caching NUMA — система NUMA без кэширования). При наличии когерентных кэш-модулей система называется **CC-NUMA** (Cache-Coherent NUMA — система NUMA с когерентным кэшированием).

Наиболее популярным подходом при построении больших мультипроцессоров CC-NUMA в настоящий момент является **каталоговый мультипроцессор**. Его идея состоит в поддержании базы данных, в которой содержится информация о том, где располагается каждая строка кэша и ее состояние. При обращении к строке кэша база данных получает запрос на поиск этой строки и выдает ее состояние («чистая» или «грязная»). Поскольку запросы этой базе данных направляются на каждой команде процессора, обращающейся к памяти, эта база должна храниться в крайне быстром специальном аппаратном устройстве, способном выдавать ответ за долю цикла шины.

Чтобы сделать идею каталоговых мультипроцессоров более понятной, рассмотрим простой (гипотетический) пример системы, состоящей из 256 узлов, каждый из которых содержит один центральный процессор и 16 Мбайт оперативной памяти, соединенной с центральным процессором локальной шиной. Общий объем ОЗУ составляет 2^{32} байт (4 Гбайт), разделенных на 2^{26} линий кэша по 64 байт каждый. Эта память статически распределяется между узлами, с адресами от 0 до 16 М в узле 0, от 16 до 32 М в узле 1 и т. д. Узлы соединяются сетью (рис. 8.6, *а*). В каждом узле также располагаются записи каталога для 2^{18} 64-байтовых линий кэша, составляющих 2^{24} байт памяти. Пока что мы будем предполагать, что каждая строка может содержаться максимум в одном кэше.

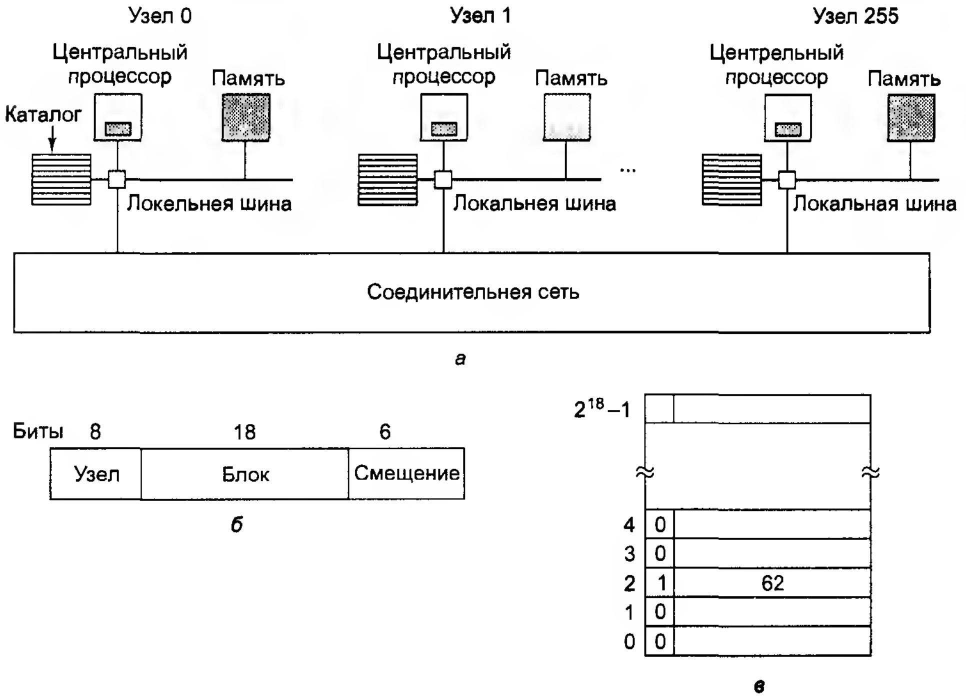


Рис. 8.6. Каталоговый мультипроцессор о 256 узлами (*а*); разделение 32-разрядного адреса на поля (*б*); каталог в узле 36 (*в*)

Чтобы понять, как работает каталог, рассмотрим выполнение команды LOAD центрального процессора 20 с обращением к строке кэша. Сначала центральный процессор, издающий команду LOAD, передает ее аргумент своему диспетчеру памяти, который преобразует его в физический адрес, например 0x24000108. Диспетчер памяти расщепляет этот адрес на три части, показанные на рис. 8.6, *б*. В десятичном виде эти части выглядят как узел 36, строка 4 и смещение 8. Диспетчер памяти видит, что центральный процессор обращается к слову памяти в узле 36, а не в узле 20, поэтому он посылает узлу 36 по соединительной сети сообщение с запросом, находится ли эта строка в кэше, и если да, то где.

Когда запрос приходит по соединительной сети на узел 36, он направляется к аппаратуре каталога. Это аппаратное обеспечение обращается по индексу в свою таблицу, состоящую из 2^{18} записей, по одной для каждой строки кэша, и достает из нее запись 4. Как видно на рис. 8.6, *в*, эта строка не является кэшированной, поэтому аппаратное обеспечение достает ее из локальной памяти, отправляет узлу 20 и отмечает в таблице, что теперь эта строка кэширована в узле 20.

Теперь рассмотрим пример другого запроса. На этот раз у узла 36 запрашивается строка 2. Как показано на рис. 8.6, *в*, эта строка кэширована в узле 82. При этом аппаратное обеспечение изменяет запись, отмечая, что теперь эта строка находится в узле 20, и посылает узлу 82 команду переслать эту строку узлу 20, а также пометить свою строку кэша как недействительную. Обратите внимание, что даже так называемый «мультипроцессор с общей памятью» вынужден пересылать большое количество сообщений незаметно для верхнего уровня.

Посчитаем, сколько памяти занимают каталоги. У каждого узла есть 16 Мбайт оперативной памяти и 2^{18} 9-битовых записей для учета этой памяти. Таким образом, накладные расходы на содержание каталога составляют около 9×2^{18} бит, что составляет около 1,76 % от 16 Мбайт. Эта величина не так уж велика и должна быть приемлемой (хотя для каталога должна использоваться высокоскоростная память, что увеличивает ее стоимость). Даже при 32-байтовых строках кэша накладные расходы на содержание каталога будут около 4 %. При 128-байтовых строках кэша накладные расходы не будут превышать 1 %.

Очевидное ограничение такой схемы заключается в том, что строка может быть кэширована только в одном узле. Чтобы позволить кэшировать строку одновременно в нескольких узлах, нам потребуется какой-то способ обнаружения всех этих строк чтобы, например, пометить их все как недействительные или чтобы обновить их при записи. Возможны различные варианты реализации этого, но обсуждение всех этих вариантов выходит за пределы данной книги.

Типы мультипроцессорных операционных систем

Перейдем теперь от обсуждения аппаратного обеспечения мультипроцессоров к рассмотрению программного обеспечения, в частности к обсуждению мультипроцессорных операционных систем. Возможны различные варианты организации данных систем. Ниже будут рассмотрены три из них.

Каждому центральному процессору — свою операционную систему

Простейший способ организации мультипроцессорных операционных систем состоит в том, чтобы статически разделить оперативную память по числу центральных процессоров и дать каждому центральному процессору свою собственную память с собственной копией операционной системы. В результате *n* центральных процессоров будут работать как *n* независимых компьютеров. В качестве очевидного варианта оптимизации можно позволить всем центральным процессорам совместно использовать код операционной системы и хранить только индивидуальные копии данных (рис. 8.7). Квадратики, помеченные словом Data, означают приватные данные операционной системы для каждого центрального процессора.

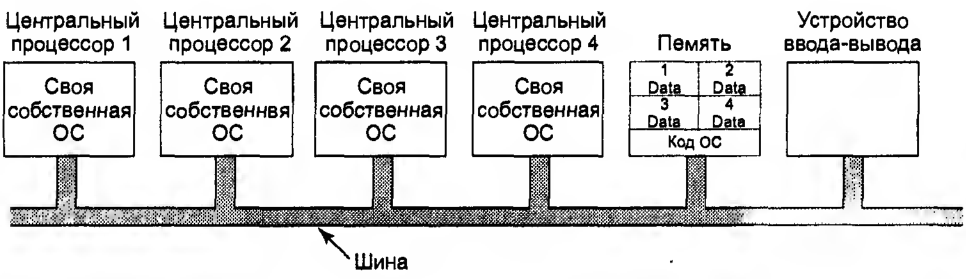


Рис. 8.7. Разделение памяти мультипроцессора между четырьмя центральными процессорами с общей копией кода операционной системы

Тем не менее такая схема лучше, чем *n* независимых компьютеров, так как она позволяет всем машинам совместно использовать набор дисков и других устройств ввода-вывода, а также обеспечивает гибкое совместное использование памяти. Например, если требуется запустить большую программу, одному из центральных процессоров может быть выделена большая порция памяти на время выполнения этой программы. Кроме того, процессы могут эффективно общаться друг с другом, если одному процессу будет позволено писать данные в память, а другой процесс будет их считывать в этом месте. Но с точки зрения операционной системы наличие операционной системы у каждого центрального процессора является крайне примитивным подходом.

Следует отметить четыре аспекта данной схемы, возможно, не являющихся очевидными. Во-первых, когда процесс обращается к системному вызову, системный вызов перехватывается и обрабатывается его собственным центральным процессором при помощи структур данных в таблицах операционной системы.

Во-вторых, поскольку у каждой операционной системы есть свои собственные таблицы, у нее есть также и свой набор процессов, которые она сама планирует. Совместного использования процессов нет. Если пользователь регистрируется на центральном процессоре 1, то все его процессы работают на центральном процессоре 1. В результате может случиться так, что центральный процессор 1 окажется загружен работой, тогда как центральный процессор 2 будет простаивать.

В-третьих, совместного использования страниц также нет. Может случиться так, что у центрального процессора 2 много свободных страниц, в то время как центральный процессор 1 будет постоянно заниматься свопингом. И нет никакого способа занять свободные страницы у соседнего процессора, так как выделение памяти статически фиксировано.

В-четвертых, и это хуже всего, если операционная система поддерживает буферный кэш недавно использованных дисковых блоков, то каждая операционная система будет выполнять это независимо от остальных. Таким образом, может случиться так, что некоторый блок диска будет присутствовать в нескольких буферах одновременно, причем в нескольких буферах сразу он может оказаться модифицированным, что приведет к порче данных на диске. Единственный способ избежать этого заключается в полном отказе от блочного кэша, что значительно снизит производительность системы.

Мультипроцессоры типа «хозяин-подчиненный»

По причине приведенных выше соображений такая модель теперь используется редко, хотя она применялась на заре эпохи мультипроцессоров, когда ставилась цель просто перенести существующие операционные системы на какой-либо новый мультипроцессор как можно быстрее. Вторая модель показана на рис 8.8. Здесь используется всего одна копия операционной системы, находящаяся на центральном процессоре 1 и отсутствующая на других центральных процессорах. Все системные вызовы перенаправляются для обработки на центральный процессор 1. Центральный процессор 1 может также выполнять процессы пользователя, если у него будет оставаться для этого время. Такая схема называется **«хозяин-подчиненный»**, так как центральный процессор 1 является «хозяином», то есть ведущим, а все остальные процессоры — подчиненными, или ведомыми.



Рис. 8.8. Модель мультипроцессора «хозяин-подчиненный»

Модель мультипроцессора «хозяин-подчиненный» позволяет решить большинство проблем первой модели. В этой модели используется единая структура данных (например, один общий список или набор приоритетных списков), учитывающая готовые процессы. Когда центральный процессор переходит в состояние простоя, он запрашивает у операционной системы процесс, который можно обрабатывать, и при наличии готовых процессов операционная система назначает этому процессору процесс. Поэтому при такой организации никогда не может случиться так, что один центральный процессор будет простаивать, в то время как другой центральный процессор перегружен. Страницы памяти могут динамически предоставляться всем процессам. Кроме того, в такой системе есть всего один общий буферный кэш блочных устройств, поэтому дискам не грозит порча данных, как в предыдущей модели при попытке использования блочного кэша.

Недостаток этой модели состоит в том, что при большом количестве центральных процессоров хозяин может стать узким местом системы. Ведь ему приходится обрабатывать все системные вызовы от всех центральных процессоров. Например, если обработка системных вызовов занимает 10 % времени, тогда 10 центральных процессоров завалят хозяина работой, а при 20 центральных процессорах хозяин уже не будет успевать их обрабатывать, и система начнет простаивать. Следовательно, такая модель проста и работоспособна для небольших мультипроцессоров, но на больших она работать не может.

Симметричные мультипроцессоры

Третья модель, представляющая собой **симметричные мультипроцессоры** (SMP, Symmetric MultiProcessor), позволяет устранить перекос предыдущей модели. Как и в предыдущей схеме, в памяти находится всего одна копия операционной системы, но выполнять ее может любой процессор. При системном вызове на центральном процессоре, обратившемся к системе с системным вызовом, происходит прерывание с переходом в режим ядра и обработкой системного вызова. Модель симметричного мультипроцессора показана на рис. 8.9.

Эта модель обеспечивает динамический баланс процессов и памяти, поскольку в ней имеется всего один набор таблиц операционной системы. Она также позволяет избежать простоя системы, связанного с перегрузкой ведущего центрального процессора («хозяина»), так как в ней нет ведущего центрального процессора. И все же данная модель имеет собственные проблемы. В частности, если код операционной системы будет выполняться одновременно на двух или более централь-

ных процессорах, произойдет катастрофа. Представьте себе два центральных процессора, одновременно берущих один и тот же процесс для запуска или запрашивающих одну и ту же свободную страницу памяти. Простейший способ разрешения подобных проблем заключается в связывании мьютекса (то есть блокировки) с операционной системой, в результате чего вся система превращается в одну большую критическую область. Когда центральный процессор хочет выполнять код операционной системы, он должен сначала получить мьютекс. Если мьютекс заблокирован, процессор вынужден ждать. Таким образом, любой центральный процессор может выполнить код операционной системы, но в каждый момент времени только один из них будет делать это.

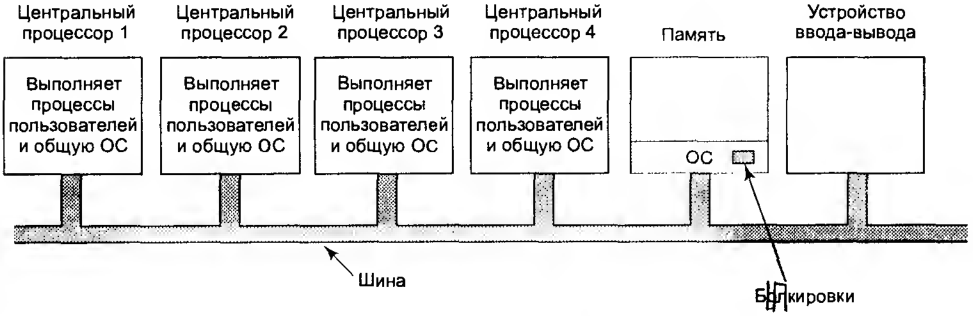


Рис. 8.9. Модель симметричного мультипроцессора

Такая модель работает, но она практически так же плоха, как и модель «хозяин-подчиненный». Опять же предположим, что 10 % всего процессорного времени расходуется на выполнение самой операционной системы. При 20 центральных процессорах большинство процессоров будут подолгу стоять в длинных очередях, ожидая разрешения доступа к мьютексу. К счастью, такую ситуацию легко исправить. Многие части операционной системы независимы друг от друга. Например, один центральный процессор может заниматься планированием, в то время как другой центральный процессор будет выполнять обращение к файловой системе, а третий обрабатывать страничное прерывание.

Такое наблюдение приводит к расщеплению операционной системы на независимые критические области, не взаимодействующие друг с другом. У каждой критической области есть свой мьютекс, поэтому только один центральный процессор может выполнять ее в каждый момент времени. Таким образом можно достичь большей степени распараллеливания. Однако может случиться, что некоторые таблицы, например таблица процессов, используются в нескольких критических областях. Так, таблица процессов требуется для планирования, но также для выполнения системного вызова fork и для обработки сигналов. Для каждой таблицы, использующейся несколькими критическими областями, требуется свой собственный мьютекс. Итак, в каждый момент времени каждая критическая область может выполняться только одним центральным процессором, а также к каждой критической таблице может быть предоставлен доступ только одному центральному процессору.

Подобная организация используется в большинстве современных мультипроцессоров. Сложность написания операционной системы для такой машины за-

ключается не в том, что программный код сильно отличается от обычной операционной системы. Нет. Самым сложным является расщепление операционной системы на критические области, которые могут выполняться параллельно на разных центральных процессорах, не мешая друг другу даже косвенно. Кроме того, каждая таблица, используемая двумя и более критическими областями, должна быть отдельно защищена мьютексом, а все программы, пользующиеся этой таблицей, должны корректно использовать мьютекс.

Кроме того, следует уделить особое внимание вопросу избежания взаимоблокировок. Если двум критическим областям одновременно потребуются таблица *A* и таблица *B* и они затребуют эти таблицы в разном порядке, то рано или поздно возникнет взаимоблокировка, причину появления которой будет очень трудно определить. Теоретически всем таблицам можно поставить в соответствие целые числа и потребовать от всех критических областей запрашивать эти таблицы по порядку номеров. Такая стратегия позволяет избежать взаимоблокировок, но она требует от программиста детального исследования того, какие таблицы потребуются каждой критической области, чтобы запросить их в правильном порядке.

При изменении программы со временем критической области может потребоваться новая таблица, которая не была нужна ранее. Если изменением программы занимается новый программист, не понимающий всей логики системы, он может поддаться искушению просто захватить мьютекс в тот момент, когда требуется таблица, и отпустить его, когда таблица более не нужна. Однако именно такие простые и кажущиеся разумными действия могут привести к взаимоблокировке, что с точки зрения пользователя выглядит как зависание системы. Очень не просто грамотно спроектировать систему, но поддерживать ее в правильном состоянии в течение нескольких лет и при этом совершенствовать систему меняющимся штатом программистов крайне трудно.

Синхронизация в мультипроцессорах

В мультипроцессорных системах часто требуется синхронизация центральных процессоров. Мы только что рассмотрели случай, в котором критические области и таблицы нуждались в защите при помощи мьютексов. Рассмотрим теперь более детально, как эта синхронизация работает в мультипроцессоре. Как мы вскоре увидим, это далеко не тривиально.

Для начала нам потребуются соответствующие примитивы синхронизации. Если процесс на однопроцессорной системе обращается к системному вызову, который требует доступа к некой критической таблице, программа ядра может просто запретить прерывания, прежде чем обратиться к таблице. Однако на мультипроцессоре запрет прерывания повлияет только на один центральный процессор, выполнивший команду запрета прерываний. Остальные центральные процессоры продолжат свою работу и смогут получить доступ к критической таблице. Требуется специальный мьютекс-протокол, который будет выполняться всеми центральными процессорами, чтобы гарантировать работу взаимного исключения.

Сердцем любого практического мьютекс-протокола является команда процессора, позволяющая исследовать и изменить слово в памяти за одну операцию. Пример использования команды TSL (Test and Set Lock — проверить и установить блоки-

ровку) для реализации критических областей был рассмотрен в листинге 2.2. Как уже было сказано, эта команда считывает слово памяти в регистр процессора. Одновременно она записывает 1 (или другое ненулевое значение) в слово памяти. Конечно, для выполнения операций чтения и записи памяти требуется два цикла шины. На однопроцессорной машине, поскольку одна команда процессора не может быть прервана на полпути, команда TSL работает должным образом.

Теперь представьте себе, что может произойти на мультипроцессоре. На рис. 8.10 показан наихудший случай совпадения по времени обращений к слову памяти по адресу 1000, начальное состояние которого 0. На шаге 1 центральный процессор 1 считывает слово и получает 0. На шаге 2, прежде чем центральный процессор 1 успевает изменить это слово, центральный процессор 2 также считывает слово и тоже получает 0. На шаге 3 центральный процессор 1 записывает в это слово 1. На шаге 4 центральный процессор 2 также записывает в это слово 1. Оба центральных процессора получили от команды TSL 0, поэтому оба считают себя вправе получить доступ к критической области. Таким образом, взаимное исключение не срабатывает.

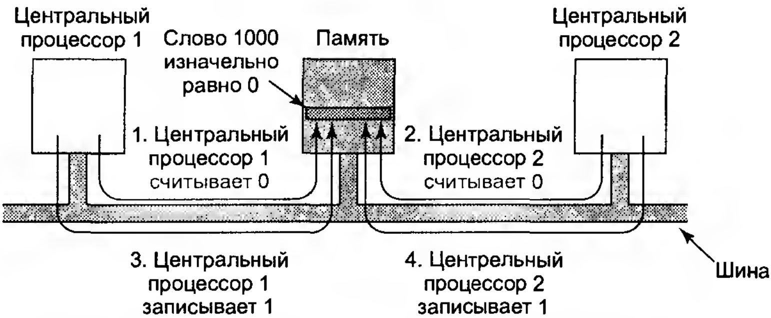


Рис. 8.10. Команда TSL может работать неверно, если не заблокировать шину

Для разрешения этой проблемы команда TSL сначала должна блокировать шину, не допуская обращения к ней других центральных процессоров, затем выполнить оба обращения к памяти, после чего разблокировать шину. Как правило, для блокировки шины сначала выполняется обычный запрос шины по стандартному протоколу, затем устанавливается в 1 некая специальная линия шины. Пока эта линия шины установлена в 1, никакой другой центральный процессор не может получить к ней доступ. Такая команда может быть выполнена только на шине, у которой есть необходимые специальные линии и (аппаратный) протокол для их использования. Современные шины обладают подобными свойствами, но на старых шинах это было невозможно, и команда TSL не могла быть выполнена корректно. Поэтому для чисто программной синхронизации был разработан протокол Петерсона [263].

Если команда TSL корректно реализована и применяется, она гарантирует правильную работу взаимного исключения. Однако подобный способ реализации взаимного исключения использует **спин-блокировку**, так как запрашивающий центральный процессор находится в цикле опроса блокировки с максимальной скоростью. Этот метод является не только тратой времени запрашивающего центрального процессора (или процессоров), но он также может накладывать значи-

тельную нагрузку на шину или память, существенно снижая скорость работы всех остальных центральных процессоров, пытающихся выполнять свою обычную работу.

На первый взгляд может показаться, что наличие кэширования должно устранить проблему конкуренции за шину, но это не так. Теоретически, как только запрашивающий центральный процессор прочитал слово блокировки, он должен получить его копию в свой кэш. Пока ни один другой центральный процессор не предпринимает попыток использовать это слово, запрашивающий центральный процессор может работать с ним в своем кэше. Когда центральный процессор, владеющий словом блокировки, записывает в него 1, протокол кэша автоматически помечает как недействительные все копии этого слова в удаленных кэшах, требуя получения правильных значений.

Проблема заключается в том, что кэш оперирует блоками по 32 или 64 байт. Обычно слова, окружающие слово блокировки, нужны центральному процессору, удерживающему это слово. Поскольку команда TSL представляет собой запись (так как она модифицирует слово блокировки), ей требуется монопольный доступ к блоку кэша, содержащему слово блокировки. Таким образом, каждая команда TSL помечает блок кэша владельца блокировки как недействительный и получает приватную, эксклюзивную копию для запрашивающего центрального процессора. Как только владелец блокировки изменит слово, соседнее с блокировкой, блок кэша перемещается на его машину. В результате весь блок кэша, содержащий слово блокировки, постоянно как челнок мотается взад-вперед от центрального процессора, удерживающего блокировку, к центральному процессору, пытающемуся ее получить. Все это создает довольно значительный и совершенно излишний трафик шины.

Проблему можно было бы решить, если бы удалось избавиться от всех операций записи, вызванных командой TSL запрашивающей стороны. Это можно сделать, если запрашивающая сторона сначала будет выполнять простую операцию чтения, чтобы убедиться, что мьютекс свободен. Только убедившись, что он свободен, центральный процессор выполняет команду TSL, чтобы захватить его. В результате большинство операций опроса представляют собой операции чтения, а не операции записи. Когда мьютекс считан, владелец выполняет операцию записи, для чего требуется монопольный доступ. При этом все остальные копии этого блока кэша объявляются недействительными. При следующей операции чтения запрашивающий центральный процессор перезагрузит этот блок кэша. Обратите внимание, что при одновременном споре двух центральных процессоров за один и тот же мьютекс может случиться, что они одновременно увидят, что мьютекс свободен, и одновременно выполнят команду TSL. Ничего страшного в этом случае не произойдет, так как выполнена будет только одна команда TSL и такая ситуация не приведет к состоянию состязания. Если вы видите, что мьютекс свободен, и тут же пытаетесь его схватить командой TSL, то нет никакой гарантии успеха данного предприятия. Мьютекс может успеть захватить кто-либо другой.

Другой способ снижения шинного трафика заключается в использовании алгоритма двоичного экспоненциального отката, применяемого в сети Ethernet [11]. Вместо постоянного опроса, показанного в листинге 2.2, между опросами может быть вставлен цикл задержки. Вначале задержка представляет собой одну команду.

Если мьютекс занят, задержка удваивается, учетверяется и т. д. до некоторого максимального уровня. При низком уровне мы получим быструю реакцию при освобождении мьютекса, зато потребуется больше обращений к шине для чтения блока кэша. Высокий максимальный уровень позволит уменьшить число лишних обращений к шине за счет более медленной реакции программы на освобождение мьютекса. Использование алгоритма двоичного экспоненциального отката не зависит от применения операций чистого чтения перед командой TSL.

Еще более удачная идея заключается в том, чтобы каждому центральному процессору, желающему заполучить мьютекс, позволить опрашивать свою собственную переменную блокировки (рис. 8.11) [233]. Во избежание конфликтов переменная должна располагаться в не используемом для других целей блоке кэша. Работа алгоритма состоит в том, что центральный процессор, которому не удается заполучить мьютекс, захватывает переменную блокировки и присоединяется к концу списка центральных процессоров, ожидающих освобождения мьютекса. Когда центральный процессор, удерживающий мьютекс, покидает критическую область, он освобождает приватную переменную, проверяемую первым центральным процессором в списке (в его собственном кэше). Тем самым следующий центральный процессор получает возможность войти в критическую область. Покинув критическую область, этот процессор освобождает переменную блокировки, тестируемую следующим процессором и т. д. Хотя такой протокол несколько сложен (это необходимо, чтобы не допустить одновременного присоединения двух центральных процессоров к концу списка), он эффективен и не страдает от проблемы голодания. Подробнее см. указанную выше статью.

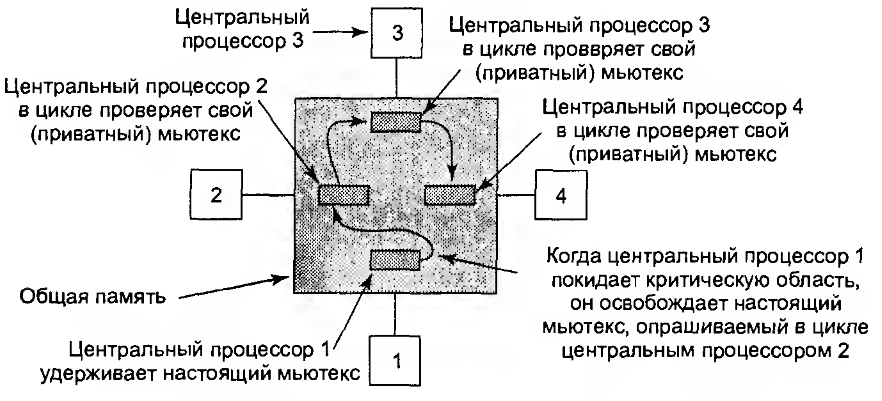


Рис. 8.11. Использование нескольких мьютексов во избежание пробуксовки кэша

Ожидание в цикле или переключения?

До сих пор мы предполагали, что центральный процессор, которому требуется мьютекс, просто ждет, пока тот не освободится, опрашивая его состояние постоянно или периодически, либо присоединяясь к списку ожидающих процессоров. В некоторых случая альтернативы простому ожиданию нет. Например, предположим, что какой-либо центральный процессор простаив ает и ему нужен доступ к общему списку готовых процессов, чтобы выбрать оттуда процесс и запустить его.

Если список готовых процессов заблокирован, простаивающему центральному процессору будет просто более нечем себя занять. Он *должен* ждать, пока список готовых процессов не освободится.

Однако в некоторых случаях у процессора может быть выбор. Например, если какому-либо потоку на центральном процессоре потребуется доступ к блочному кэшу файловой системы, заблокированному в данный момент, центральный процессор может решить не ждать, а переключиться на другой поток. Вопрос принятия решения о том, ждать или переключиться на другой поток, являлся предметом многочисленных исследований, некоторые из них будут обсуждаться ниже. Обратите внимание, что на однопроцессорной системе такого вопроса не возникает, так как опрос в цикле ни имеет смысла при отсутствии другого центрального процессора, способного освободить мьютекс. Если поток пытается получить мьютекс и ему это не удается, он всегда блокируется, чтобы дать возможность владельцу мьютекса работать и освободить мьютекс.

Если допустить, что оба варианта (опрос в цикле и переключение потоков) выполнимы, возникает следующая ситуация. Циклический опрос напрямую расходует процессорное время. Периодическая проверка состояния мьютекса не является продуктивной работой. Переключение процессов, однако, также расходует циклы процессора, так как для этого требуется сохранить текущее состояние потока, получить мьютекс списка свободных процессов, выбрать из этого списка поток, загрузить его состояние и передать ему управление. Более того, кэш центрального процессора будет содержать не те блоки, поэтому после переключения потоков возможно много промахов кэша. Также вероятны ошибки TLB. Наконец, потребуется переключение обратно на исходный поток, результатом которого снова будут промахи кэша. Циклы процессора, потраченные на эти два переключения контекста плюс промахи кэша, представляют собой существенные накладные расходы.

Если известно, что мьютексы удерживаются, как правило, в течение 50 мкс, а переключение с одного потока на другой занимает 1 мс, а также 1 мс требует обратное переключение, то более эффективное решение состоит в простом ожидании освобождения мьютекса в цикле. С другой стороны, если средний мьютекс удерживается на 10 мс, то два переключения контекста являются вполне оправданными. Проблема в том, что длительность нахождения процессов в критических областях может варьироваться в широких пределах, поэтому выбор верного решения непрост.

Одно решение заключается в том, чтобы всегда использовать циклический опрос. Вторым подходом является использование только переключений. Третий вариант состоит в том, чтобы каждый раз принимать отдельное решение. В момент принятия решения не известно, что лучше — переключаться или ждать, но в любой системе можно отследить активность процессов и затем проанализировать ее в автономном режиме. При этом можно сказать, какое решение было бы лучшим в том или ином случае и сколько процессорного времени было потрачено. Таким образом, ретроспективный алгоритм становится эталоном, относительно которого может измеряться эффективность реальных алгоритмов.

Эта проблема уже изучалась исследователями [173, 172, 255]. В большинстве работ используется модель, в которой поток, не заполучивший мьютекс, какое-то время опрашивает состояние мьютекса в цикле. Если пороговое значение времени

ожидания превышается, он переключается. В некоторых случаях пороговое значение фиксировано, как правило, оно равно известному времени, требующемуся на переключение на другой поток и обратно. В других случаях эта величина меняется динамически, в зависимости от истории состояния ожидаемого мьютекса.

Лучшие результаты достигались тогда, когда система учитывала несколько последних интервалов ожидания и предполагала, что следующий интервал ожидания будет близок по величине к предыдущим. Например, снова предполагая, что переключение контекста занимает 1 мс, поток будет ожидать в цикле максимум 2 мс, но запоминать при этом, сколько времени он на самом деле провел в состоянии ожидания. Если ему не удается захватить мьютекс и он видит, что за предыдущие три попытки он ждал в среднем 200 мкс, значит, ему следует ждать 2 мс, прежде чем переключаться. Однако если он находился в цикле ожидания целые 2 мс в каждую из своих предыдущих попыток, то он должен переключиться немедленно, не тратя на ожидание ни одного цикла. Дополнительные детали можно найти в [172].

Планирование мультипроцессора

На однопроцессорной системе планирование одномерно. Единственный вопрос, на который должен быть каждый раз получен ответ, — какой процесс должен быть запущен следующим? На мультипроцессоре планирование двумерно. Планировщик должен решить, какой процесс и на котором центральном процессоре запустить. Это дополнительное измерение существенно усложняет планирование на мультипроцессорах.

Другой усложняющий фактор состоит в том, что в некоторых системах все процессы являются независимыми, тогда как в других системах они формируют группы. Примером первой ситуации является система реального времени, в которой независимые пользователи запускают независимые процессы. Эти процессы не связаны друг с другом, и планирование каждого из них не зависит от остальных процессов.

Пример второй ситуации часто встречается в среде разработки программ. Большие системы, как правило, состоят из некоторого количества заголовочных файлов, содержащих макросы, определения типов и объявления переменных, используемых в собственно файлах программы. При изменении заголовочного файла все программные файлы, в которые он включен, должны быть перекомпилированы. Для этого обычно используется программа *make*. При вызове программа *make* начинает компиляцию только тех файлов, которые должны быть перекомпилированы из-за изменений самих файлов или заголовочных файлов, включенных в эти файлы кода. Все еще имеющие силу объектные файлы заново не пересоздаются.

Оригинальная версия программы *make* выполняла свою работу последовательно, но новые версии, созданные для мультипроцессоров, могут начинать всю компиляцию одновременно. Если требуется откомпилировать 10 файлов, то нет смысла выполнить компиляцию 9 файлов быстро, а последний файл отложить на потом, так как пользователь будет считать работу выполненной только по завершении компиляции последнего файла. В этом случае есть смысл рассматривать все эти процессы как группу и учитывать это при их планировании.

Разделение времени

Рассмотрим сначала случай планирования независимых процессов. Затем мы обсудим, как планировать зависимые процессы. Простейший алгоритм планирования независимых процессов (или потоков) состоит в поддержании единой структуры данных для готовых процессов, возможно, просто списка, но скорее всего множества списков для процессов с различными приоритетами (рис. 8.12, *а*). Здесь все 16 центральных процессоров в данный момент заняты, а 14 процессов с различными приоритетами ожидают запуска. Первым заканчивает работу (или его процесс блокируется) центральный процессор 4. При этом он блокирует очередь планирования и выбирает из нее процесс с наивысшим приоритетом, то есть процесс *A* (рис. 8.12, *б*). Затем освобождает центральный процессор 12 и выбирает процесс *B* (рис. 8.12, *в*). Пока эти процессы независимы, подобное планирование представляет собой разумный выбор.

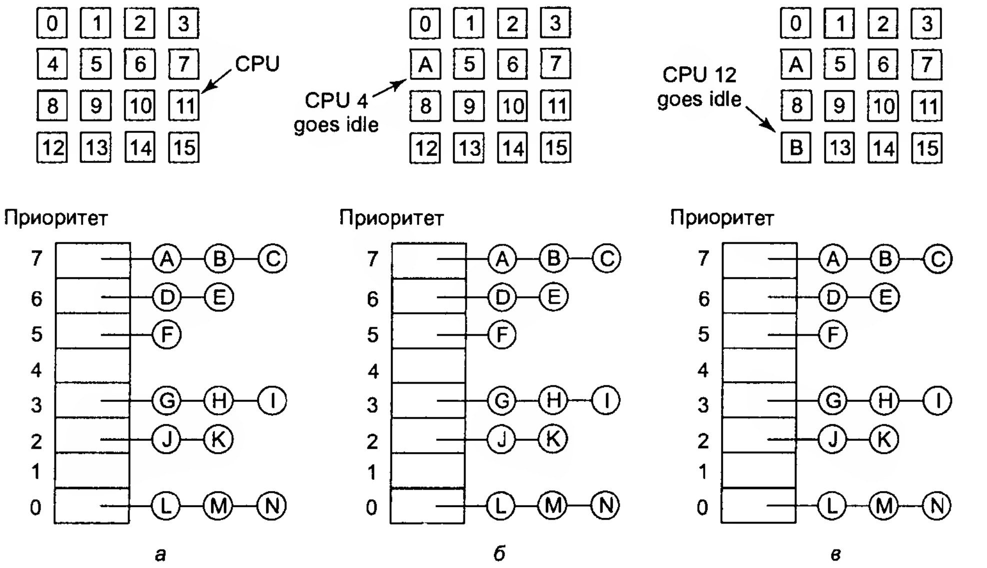


Рис. 8.12. Использование единой структуры данных для планирования мультипроцессора

Наличие единой структуры данных планирования, используемой всеми центральными процессорами, обеспечивает процессорам режим разделения времени подобно тому, как это выполняется на однопроцессорной системе. Кроме того, такая организация позволяет автоматически балансировать нагрузку, то есть она исключает ситуацию, при которой один центральный процессор простаивает, в то время как другие процессоры перегружены. Два недостатка такой схемы представляют собой потенциальный рост конкуренции за структуру данных планирования по мере увеличения числа центральных процессоров и обычные накладные расходы на выполнение переключения контекста, когда процесс блокируется, ожидая выполнения операции ввода-вывода.

Переключение контекста также может случиться, когда истекает квант процесса. Мультипроцессор обладает определенными свойствами, которыми не обладают однопроцессорные системы. Предположим, что процесс удерживает спин-

блокировку, что, как уже говорилось выше, часто встречается на мультипроцессорах. Другие центральные процессоры, ждущие освобождения блокировки, просто теряют время в циклах ожидания, пока этот процесс не будет запущен снова и не отпустит блокировку. На однопроцессорных системах спин-блокировка применяется редко. Поэтому если процесс, удерживающий мьютекс, блокируется и запускается другой процесс, то при попытке заполучить мьютекс второй процесс будет тут же блокирован, и много времени потеряно не будет.

Чтобы решить данную проблему, в некоторых системах применяется **умное планирование**, в котором процесс, захватывающий спин-блокировку, устанавливает флаг, демонстрирующий, что он в данный момент обладает спин-блокировкой [368]. Когда процесс освобождает блокировку, он также очищает и флаг. Таким образом, планировщик не останавливает процесс, удерживающий спин-блокировку, а, напротив, дает ему еще немного времени, чтобы тот завершил выполнение критической области и отпустил мьютекс.

Другая проблема, играющая важную роль в планировании, заключается в том факте, что хотя все центральные процессоры равны, но некоторые центральные процессоры равнее других. В частности, когда процесс *A* достаточно долго работает на центральном процессоре *k*, кэш процессора *k* будет полон блоками процесса *A*. Если процесс *A* должен быть снова вскоре запущен, его производительность может оказаться выше, если он будет запущен снова на центральном процессоре *k*, так как кэш процессора *k* может все еще содержать некоторые блоки процесса *A*. Наличие блоков в кэше увеличит частоту попаданий кэша и, таким образом, увеличит скорость выполнения процесса. Кроме того, TLB также может содержать правильные страницы, что снизит количество прерываний из-за ошибок TLB.

Некоторые мультипроцессоры учитывают данные соображения и используют так называемое **родственное планирование** [342]. Основная идея данного метода заключается в приложении серьезных усилий для того, чтобы процесс был запущен на том же центральном процессоре, что и в прошлый раз. Один из способов реализации этого метода состоит в использовании **двухуровневого алгоритма планирования**. В момент создания процесс назначается конкретному центральному процессору, например наименее загруженному в данный момент. Это назначение процессов процессорам представляет собой верхний уровень алгоритма. В результате каждый центральный процессор получает свой набор процессов.

Действительное планирование процессов находится на нижнем уровне алгоритма. Оно выполняется отдельно каждым центральным процессором при помощи приоритетов или других средств. Старания удерживать процессы на одном и том же центральном процессоре максимизируют родственность кэша. Однако если у какого-либо центрального процессора нет работы, у загруженного работой процессора отнимается процесс и отдается ему.

Двухуровневое планирование обладает тремя преимуществами. Во-первых, оно довольно равномерно распределяет нагрузку среди имеющихся центральных процессоров. Во-вторых, двухуровневое планирование по возможности использует преимущество родственности кэша. В-третьих, поскольку у каждого центрального процессора при таком варианте планирования есть свой собственный список свободных процессов, конкуренция за списки свободных процессов минимизируется, так как попытки использования списка другого процессора происходят относительно нечасто.

Совместное использование пространства

Другой подход к планированию мультипроцессоров может быть использован, если процессы связаны друг с другом каким-либо способом. Ранее уже обсуждался пример параллельного выполнения процедуры *make*. Часто также случается, что один процесс создает множество потоков, работающих совместно. Для нашей задачи задание, состоящее из нескольких связанных процессов или процесс, состоящий из нескольких потоков ядра, представляют собой, по сути, одно и то же. Мы будем называть планируемые объекты потоками, но все сказанное здесь в равной мере справедливо и для процессов. Планирование нескольких потоков на нескольких центральных процессорах называется **совместным использованием пространства** или **разделением пространства**.

Простейший алгоритм разделения пространства работает следующим образом. Предположим, что сразу создается целая группа связанных потоков. В момент их создания планировщик проверяет, есть ли свободные центральные процессоры по количеству создаваемых потоков. Если свободных процессоров достаточно, каждому потоку выделяется собственный (то есть работающий в однозадачном режиме) центральный процессор и все потоки запускаются. Если процессоров недостаточно, ни один из потоков не запускается, пока не освободится достаточное количество центральных процессоров. Каждый поток выполняется на своем процессоре вплоть до завершения, после чего все центральные процессоры возвращаются в пул свободных процессоров. Если поток оказывается заблокированным операцией ввода-вывода, он продолжает удерживать центральный процессор, который простаивает до тех пор, пока поток не сможет продолжать свою работу. При появлении следующего пакета потоков применяется тот же алгоритм.

В любой момент времени множество центральных процессоров статически разделяется на несколько подмножеств, на каждом из которых выполняются потоки одного процесса. На рис. 8.13 показаны подмножества из 4, 6, 8 и 12 процессоров, и 2 процессора остались не включенными в подмножества. Со временем, по мере завершения работы одних процессов и появления новых процессов, количество и размеры групп центральных процессоров изменяются.



Рис. 8.13. Набор из 32 центральных процессоров, разделенный на четыре группы, плюс два процессора свободны

Периодически должны приниматься решения о планировании процессов. В однопроцессорных системах для пакетного планирования применяется хорошо изве-

стный алгоритм «кратчайшее задание первое». Подобный алгоритм для мультипроцессора представляет собой выбор процесса, для которого требуется наименьшее количество циклов процессора, то есть процесса, чье произведение числа центральных процессоров на время работы минимально. Однако на практике эта информация редко бывает доступна, поэтому применение такого алгоритма затруднительно. Действительно, исследования показали, что придумать что-либо лучшее простого обслуживания заданий в порядке их поступления трудно [188].

В этой простой модели разбиения процессоров на группы процесс просто запрашивает определенное количество центральных процессоров и либо сразу получает их, либо ждет, пока они не освободятся. Другой подход состоит в том, чтобы активно управлять степенью распараллеливания процессов. Один из способов управления степенью распараллеливания заключается в наличии центрального сервера, ведущего учет работающих и желающих работать процессов, а также минимального и максимального количества требующихся для них центральных процессоров [332]. Периодически каждый центральный процессор опрашивает центральный сервер, чтобы узнать, сколько центральных процессоров он может использовать. Затем он увеличивает или уменьшает количество процессов или потоков, стараясь добиться соответствия числу доступных процессоров. Например, на web-сервере могут параллельно работать 1, 2, 5, 10, 20 или любое другое количество потоков. Если на нем в настоящий момент работает 10 потоков и вдруг спрос на центральные процессоры повышается, то ему могут приказать сократить число своих потоков до 5. Поэтому, когда 5 потоков закончат свою работу, они не получают новую работу, а завершаются. Такая схема обеспечивает динамическое изменение размеров групп процессоров, чтобы добиться лучшего соответствия текущей нагрузке, нежели фиксированная система на рис. 8.13.

Бригадное планирование

Очевидное преимущество разделения пространства заключается в устранении многозадачности, что снижает накладные расходы по переключению контекста. Однако ее недостаток состоит в потерях времени при блокировке центрального процессора. Поэтому проводились активные поиски алгоритма, пытающегося планировать одновременно в пространстве и времени, особенно для процессов, создающих большое количество потоков, которым, как правило, требуется возможность общения друг с другом.

Чтобы понять, какие возможны проблемы при независимом планировании потоков процесса (или процессов задания), рассмотрим систему с потоками A_0 и A_1, принадлежащими процессу A, и потоками B_0 и B_1, принадлежащими процессу B. Потоки A_0 и B_0 работают в режиме разделения времени на центральном процессоре 0, а потоки A_1 и B_1 — на процессоре 1. Потокам A_0 и A_1 нужно часто обмениваться информацией. Общение потоков выглядит следующим образом. Поток A_0 посылает потоку A_1 сообщение, после чего поток A_1 отправляет потоку A_0 ответ и т. д. Предположим, что потоки A_0 и B_1 начали выполняться первыми, как показано на рис. 8.14.

В интервале времени 0 поток A_0 посылает потоку A_1 запрос, но поток A_1 не получает его до тех пор, пока не будет запущен в интервале времени 1, начинающем-

ся через 100 мс. Он немедленно отправляет ответ, но поток A_0 не получает ответа, пока его снова не запустят в момент времени 200 мс. В результате за 200 мс мы получаем всего одну пару запрос-ответ, что не слишком хорошо.

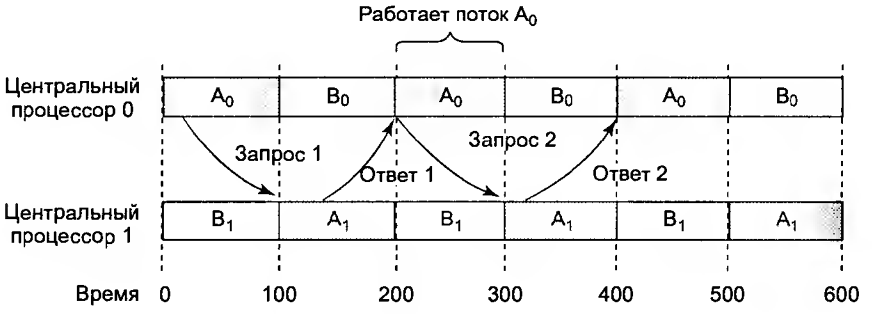


Рис. 8.14. Общение двух работающих в противофазе потоков, принадлежащих процессу *A*

Решением данной проблемы является так называемое **бригадное планирование**, представляющее собой развитие идеи **совместного планирования** [255]. Бригадное планирование состоит из трех частей:

1. Группы связанных потоков планируются как одно целое, бригада.
2. Все члены бригады запускаются одновременно, на разных центральных процессорах с разделением времени.
3. Все члены бригады начинают и завершают свои временные интервалы вместе.

Бригадное планирование работает благодаря синхронности работы всех центральных процессоров. Это значит, что время разделяется на дискретные кванты, как было показано на рис. 8.14. В начале каждого нового кванта *все* центральные процессоры перепланируются заново, и на каждом процессоре запускается новый поток. В начале следующего кванта опять принимается решение о планировании. В середине кванта планирование не выполняется. Если какой-либо поток блокируется, его центральный процессор простаивает до конца кванта времени.

Пример работы бригадного планирования приведен на рис. 8.15. Здесь показан мультипроцессор с шестью процессорами, на которых работают пять процессов, от *A* до *E*, с общим числом потоков, равным 24. В течение временного интервала 0 потоки от A_0 до A_6 планируются и выполняются. Во время интервала 1 планируются и выполняются потоки B_0, B_1, B_2, C_0, C_1, и C_2. В течение временного интервала 2 планируются и выполняются пять потоков процесса *D* и поток E_0. Оставшиеся шесть потоков процесса *E* работают во время интервала 3. Затем цикл повторяется, так что временной интервал 4 повторяет интервал 0 и т. д.

Идея бригадного планирования состоит в том, чтобы все потоки процесса работали по возможности вместе, так, что если один из них посылает сообщение другому потоку, то второй поток получает сообщение практически мгновенно и может так же быстро на него ответить. На рис. 8.15, поскольку все потоки процесса *A* работают вместе в течение одного кванта времени, они могут отправлять и принимать большое количество сообщений за один квант времени, устраняя, таким образом, проблему рис. 8.14.

Центральный процессор

Временной интервал	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Рис. 8.15. Бригадное планирование

Многомашинные системы

Привлекательность и популярность мультипроцессоров связана с тем, что они предлагают простую коммуникативную модель: все центральные процессоры просто совместно используют общую память. Процессы могут писать в память сообщения, которые затем могут считываться другими процессами. Синхронизация может выполняться при помощи мьютексов, семафоров, мониторов и других хорошо разработанных методов. Единственная ложка дегтя в бочке меда заключается в том, что большие мультипроцессоры сложны в построении и поэтому дороги.

Для решения этой проблемы большое количество исследований было посвящено **многомашинным системам** или **мультикомпьютерам**, представляющим собой тесно связанные центральные процессоры, у которых нет общей памяти. У каждого из них имеется своя отдельная оперативная память, как было показано на рис. 8.1, *б*. У этих систем есть множество других названий, включая **кластерные компьютеры** и **COWS** (Clusters of Workstations — гроздья рабочих станций).

Создание таких систем не представляет сложности, так как их основными компонентами являются усеченные варианты обычных персональных компьютеров с добавлением сетевой интерфейсной карты. Разумеется, секрет достижения высокой производительности кроется в грамотной схеме соединительной сети и интерфейсной карты. Эта проблема полностью аналогична проблеме создания общей памяти в мультипроцессоре. Однако в многомашинных системах целью является передача сообщений за микросекунды, а не доступ к памяти за наносекунды, поэтому эта задача проще и, соответственно, ее решение является более простым и дешевым в реализации.

В следующих разделах мы сначала кратко рассмотрим аппаратное обеспечение многомашинных систем, особенно аппаратное обеспечение соединительной сети. После этого мы перейдем к обсуждению программного обеспечения, сначала низкоуровневых коммуникационных программ, а затем высокоуровневого программного обеспечения связи. Мы также обсудим способы программной реализации совместно используемой памяти в системах, у которых нет аппаратной общей памяти. Наконец, мы рассмотрим вопросы планирования и балансировки нагрузки.

Аппаратное обеспечение многомашинных систем

Базовый узел многомашинной системы состоит из центрального процессора, памяти, сетевого интерфейса и иногда жесткого диска. Этот узел может быть упакован в стандартный корпус персонального компьютера, но графический адаптер, монитор, клавиатура и мышь у него практически всегда отсутствуют. В некоторых случаях такой персональный компьютер может содержать в себе вместо одного центрального процессора 2-процессорную или 4-процессорную плату, но для простоты мы будем предполагать, что у каждого узла всего один центральный процессор. Часто для создания многомашинных систем объединяются сотни или даже тысячи узлов. Ниже будет кратко рассказано о том, как организовано это аппаратное обеспечение.

Технология соединений

У каждого узла есть сетевая интерфейсная карта с выходящими из нее одним или двумя кабелями (коаксиальными или оптоволоконными). Эти кабели соединяются с другими узлами или коммутаторами. В небольшой системе может существовать один коммутатор, к которому присоединяются все узлы, образуя топологию «звезда» (рис. 8.16, *а*). Подобная топология применяется в современных коммутируемых сетях Ethernet.

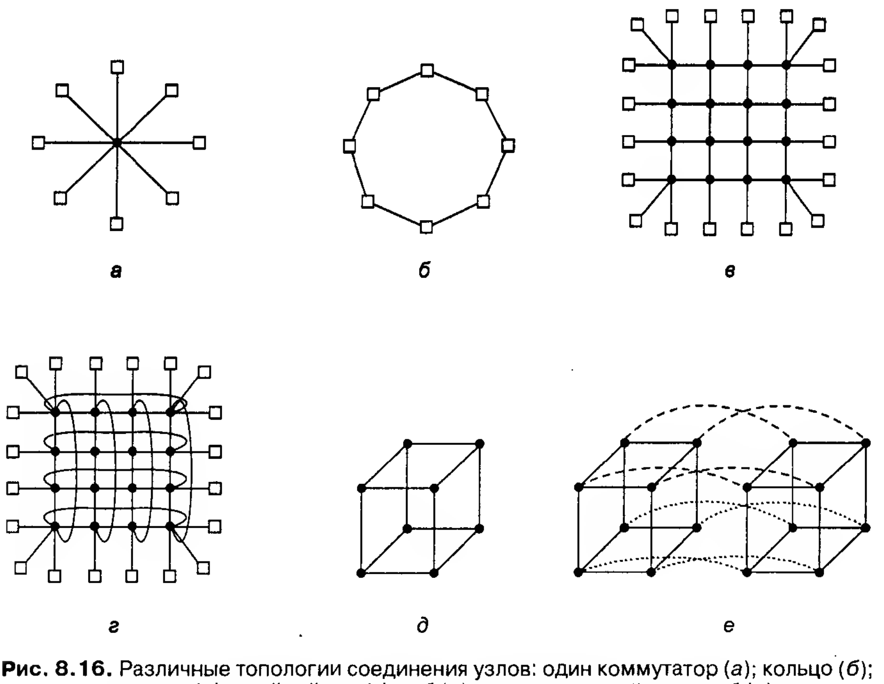


Рис. 8.16. Различные топологии соединения узлов: один коммутатор (*а*); кольцо (*б*); решетка (*в*); двойной тор (*г*); куб (*д*); четырехмерный гиперкуб (*е*)

Альтернативная топология может представлять собой кольцо, в котором каждый узел соединяется с двумя соседними узлами без помощи коммутаторов (рис. 8.16, *б*).

Третий вариант топологии представляет собой **решетку** или **сеть** (рис. 8.16, *в*). Это уже двумерная топология, применяемая во многих коммерческих системах. Она обладает высокой степенью регулярности и легко масштабируется до больших размеров. Топология решетки имеет **диаметр**, которым называют самый длинный путь между двумя узлами. Диаметр решетки увеличивается пропорционально квадратному корню от общего числа узлов решетки. Один из вариантов топологии решетки, у которой крайние узлы соединены друг с другом, называется **двойным тором** (рис. 8.16, *г*). Такая схема обладает большей устойчивостью к повреждениям и сбоям, чем простая решетка, к тому же дополнительные линии связи снижают ее диаметр.

Топология **куб** (рис. 8.16, *д*) представляет собой регулярную трехмерную топологию. На рисунке изображен куб размером 2×2×2, но на практике применяются кубы значительно больших размеров. На рис. 8.16, *е* показан четырехмерный куб, созданный из двух трехмерных кубов с помощью соединений соответствующих узлов. Эту идею можно развивать, создавая пяти- и шестимерные кубы и т. д. Созданный таким образом n-мерный куб называется **гиперкубом**. Подобная топология используется во многих параллельных компьютерах, потому что диаметр в них растет линейно в зависимости от размерности. Другими словами, диаметр представляет собой логарифм по основанию 2 от числа узлов, например, у 10-мерного гиперкуба с 1024 узлами диаметр будет равен всего 10, в результате чего такая схема обладает прекрасными временными характеристиками (низкой задержкой). Обратите внимание, что если организовать эти 1024 узла в виде квадратной решетки 32×32, то диаметр в этом случае будет равен 62, что более чем в шесть раз хуже, чем для гиперкуба. Платой за небольшой диаметр гиперкуба является большое число ответвлений на каждом узле, пропорциональное размерности гиперкуба, и, таким образом, большее количество (и большая стоимость) связей между узлами.

В многомашинных системах применяются две коммутационные схемы. В первой из них каждое сообщение сначала разбивается (либо программным обеспечением пользователя, либо сетевым интерфейсом) на отдельные фрагменты, называемые **пакетами**. Коммутационная схема, называемая **коммутацией пакетов с промежуточным хранением**, состоит в том, что пакет сначала посылается сетевой картой источника первому узлу (рис. 8.17, *а*). Пакет передается побитно, и когда прибывает весь пакет, он копируется на следующий коммутатор (рис. 8.17, *б*). Когда пакет прибывает на коммутатор, соединенный с пунктом назначения (рис. 8.17, *в*), пакет копируется на сетевую карту узла-получателя и, наконец, попадает в оперативную память этого узла.

Хотя схема коммутации пакетов с промежуточным хранением обладает гибкостью и эффективностью, у нее есть проблема, заключающаяся в увеличении задержки передачи сообщения по сети. Предположим, что время передачи одного пакета на один транзитный участок на рис. 8.17 составляет T нс. Поскольку для передачи пакета от центрального процессора 1 до центрального процессора 2 требуется закопировать этот пакет четыре раза (на коммутаторы A, C и D и, наконец, узлу-получателю) и ни одна операция копирования не может начаться, пока не будет завершена предыдущая, задержка передачи по соединительной сети составляет $4T$ нс. Один из способов решения данной проблемы заключается в создании гибридной сети, обладающей некоторыми свойствами коммутации пакетов и неко-

торыми свойствами коммутации каналов. Например, каждый пакет может логически разбиваться на более мелкие единицы. Как только первая такая единица прибывает на коммутатор, она может пересылаться дальше на следующий коммутатор, прежде чем прибудет остальная часть пакета.

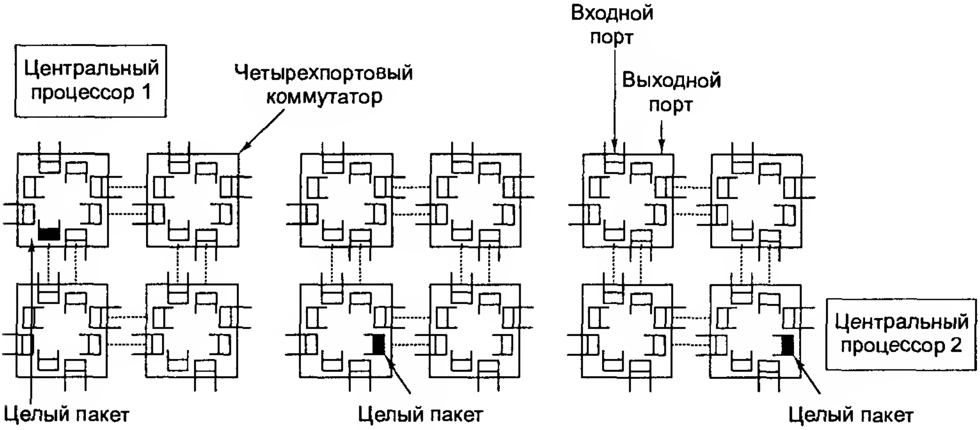


Рис. 8.17. Коммутация пакетов с промежуточным хранением

Другой режим коммутации, **коммутация каналов**, заключается в том, что первый коммутатор сначала устанавливает путь через все коммутаторы до коммутатора-получателя. Как только путь установлен, биты передаются от источника к приемнику без остановок. При этом промежуточного хранения не производится. Для коммутации каналов требуется фаза начальной установки, занимающая некоторое время, зато по завершении этапа установки весь процесс передачи данных идет быстрее. Когда сообщение отправлено, путь должен быть снова разорван. Существует вариант коммутации каналов, называемый **червячной маршрутизацией**, при которой каждый пакет разбивается на подпакеты, и первый подпакет начинает передаваться еще до того, как был установлен полный путь.

Сетевые интерфейсы

У всех узлов многомашинных систем есть плата, на которой расположены все соединения узла с соединительной сетью, удерживающей мультикомпьютер вместе. Архитектура сетевой платы и способ ее соединения с центральным процессором и оперативной памятью оказывают существенное влияние на операционную систему. В данном разделе будут кратко рассмотрены некоторые из этих вопросов. Этот материал частично основан на [29]. Кроме того, использовалась следующая литература: [49, 257, 312, 348].

Практически во всех многомашинных системах интерфейсная карта содержит некоторое количество ОЗУ для хранения входящих и выходящих пакетов. Как правило, выходной пакет должен быть скопирован на интерфейсную плату прежде, чем она сможет начать его передачу первому коммутатору. Причина этого состоит в том, что многие соединительные сети являются синхронными, поэтому, как только начинается передача одного пакета, его биты должны передаваться с постоян-

ной скоростью. Если пакет находится в оперативной памяти, постоянство потока гарантировать невозможно, поскольку на шине памяти может существовать и другой трафик. Наличие выделенной памяти в интерфейсной карте устраняет эту проблему. Схема из четырех узлов показана на рис. 8.18.

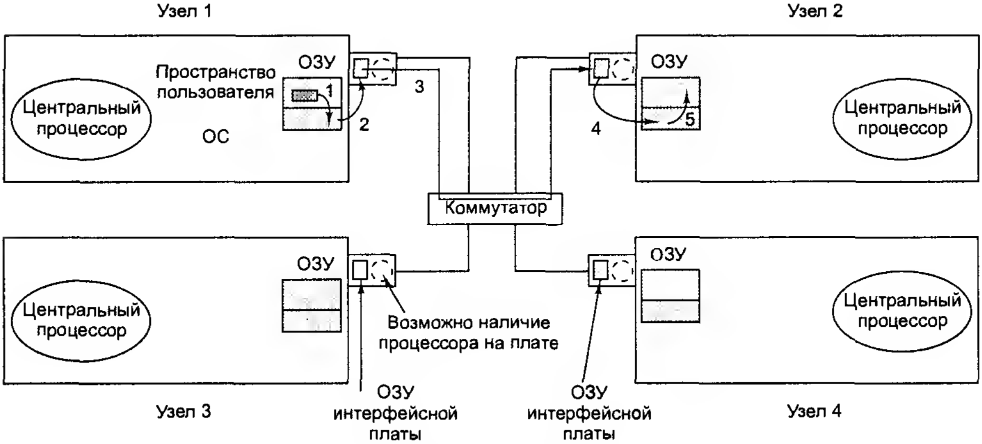


Рис. 8.18. Расположение сетевых интерфейсных плат в мультикомпьютере

Та же проблема касается приходящих пакетов. Биты прибывают по сети с постоянной и иногда очень высокой скоростью. Если сетевой интерфейс не может хранить их в режиме реального времени, по мере прибытия данные будут теряться. Попытка сразу передавать их по системной шине (например, по шине PCI) в оперативную память слишком рискованна. Поскольку сетевая карта, как правило, подключается к шине PCI, эта шина представляет собой единственную связь сетевой карты с оперативной памятью, и неизбежно соревнование за эту шину с диском и другими устройствами ввода-вывода. Спокойнее сохранять приходящие пакеты в независимом ОЗУ сетевой карты, а затем копировать их в оперативную память.

У интерфейсной платы может быть один или несколько каналов DMA или даже целый процессор на плате. Каналы DMA могут копировать пакеты между интерфейсной картой и оперативной памятью с большой скоростью, запрашивая блочную передачу по системной шине. Такой режим позволяет передавать по шине несколько слов подряд, не запрашивая шину для каждого слова отдельно. Именно для возможности использования такого режима ОЗУ на интерфейсной карте нужно в первую очередь.

На некоторых интерфейсных картах устанавливается полноценный центральный процессор, к которому, возможно, добавляются один или несколько каналов DMA. Такая схема означает, что основной центральный процессор может переложить часть нагрузки на сетевую карту, например обеспечение надежной передачи (при условии, что лежащее в основе аппаратное обеспечение может терять пакеты), реализацию многоадресной рассылки (отправка пакетов более чем одному адресату) и заботу о защите в системе с несколькими процессами. Однако нали-

чие нескольких центральных процессоров означает, что они должны синхронизироваться во избежание конфликтов, что увеличивает сложность системы и означает больше работы для операционной системы.

Коммуникационное программное обеспечение низкого уровня

Врагом высокопроизводительной связи в многомашинных системах является излишнее копирование пакетов. В лучшем случае происходит одна операция копирования из ОЗУ на интерфейсную плату источника, одна операция копирования с интерфейсной платы источника на интерфейсную плату приемника (если промежуточное хранение в сети не применяется) и одна операция копирования с интерфейсной платы приемника в оперативную память. Итого три операции копирования. Однако во многих системах ситуация оказывается еще хуже. В частности, если интерфейсная карта отображается на память в адресном пространстве ядра, а не в адресном пространстве пользователя, процесс пользователя может послать пакет, только обратившись к системному вызову, который с помощью эмулированного прерывания переключится в пространство ядра. Программам ядра, возможно, потребуется скопировать пакет в свой собственный буфер как при передаче, так и при приеме пакета, например, чтобы избежать страничных прерываний при передаче по сети. Кроме того, получающее ядро, может статься, не будет знать, где разместить пакет, пока оно не изучит содержимое пакета. Эти пять операций копирования показаны на рис. 8.18.

Если производительность главным образом зависит от копирования в ОЗУ и из ОЗУ, лишние операции копирования могут удвоить сквозную задержку и вдвое снизить пропускную способность. Во избежание такого удара по производительности во многих мультикомпьютерах применяется отображение интерфейсных карт в адресное пространство пользователя. При этом пользовательские процессы получают возможность напрямую помещать пакеты в интерфейсную плату, без обращения к ядру. Хотя такой подход определенно помогает увеличить производительность, с его применением связаны две проблемы.

Во-первых, что если на узле одновременно работает несколько процессов, которым нужен доступ к сети для отправки пакетов? Какой из них получит интерфейсную карту в свое адресное пространство? Использование системного вызова для отображения карты на виртуальное адресное пространство является дорогим удовольствием, но если один процесс получит интерфейсную карту, то как остальные процессы смогут получать и отправлять пакеты? И что произойдет, если карта будет отображена на виртуальное адресное пространство процесса *A*, а пакет прибывает для процесса *B*, особенно если у этих процессов разные владельцы и ни один из них не хочет пальцем о палец ударить, чтобы помочь другому?

Одно решение заключается в том, чтобы отобразить интерфейсную карту на все процессы, которым она нужна, но тогда потребуется специальный механизм предотвращения конфликтов. Например, если процесс *A* получит буфер интерфейсной карты, после чего, поскольку временной интервал работы процесса *A* закончится, процесс *B* также получит буфер интерфейсной карты, то результаты будут

катастрофическими. Необходим определенный механизм синхронизации доступа процессов к интерфейсной карте, но такие механизмы, как мьютексы, работают только тогда, когда предполагается сотрудничество процессов. В среде разделения времени с многочисленными пользователями, спешащими выполнить свою работу, один пользователь может просто захватить мьютекс, связанный с картой, и не отдавать его. Отсюда следует вывод, что отображение интерфейсной карты на пространство пользователя будет работать только в том случае, когда в каждом узле работает лишь по одному пользовательскому процессу или предпринимаются специальные меры (например, разным процессам предоставляются различные участки ОЗУ интерфейсной карты).

Вторая проблема состоит в том, что ядру также может потребоваться доступ к интерфейсной карте, например, для доступа к файловой системе на удаленном узле. Совместное использование интерфейсной карты ядром и пользователями представляет собой неудачную идею даже на основе разделения времени. Предположим, что в то время, пока карта отображается в пространство пользователя, прибывает пакет ядра. Или представьте, что пользовательский процесс пошлет удаленной машине пакет, притворившись ядром. Отсюда следует вывод, что простейшая схема представляет собой две интерфейсные карты, одна из которых отображается на пространство пользователя, а вторая отображается на пространство ядра и используется операционной системой. Многие многомашинные системы действуют именно так.

Связь между узлом и сетевым интерфейсом

Другой вопрос заключается в том, как пакеты попадают в интерфейсную карту. Самый быстрый способ состоит в использовании микросхемы DMA, размещенной на интерфейсной плате, чтобы просто копировать пакеты в ОЗУ. Недостаток этого метода заключается в том, что контроллер DMA использует не виртуальное, а физическое адресное пространство и работает независимо от центрального процессора. Во-первых, хотя процесс пользователя знает виртуальный адрес любого пакета, который он хочет отправить, ему, как правило, не известен физический адрес. Выполнение системного вызова для преобразования виртуального адреса в физический нежелательно, так как целью расположения интерфейсной карты в адресном пространстве пользователя была как раз попытка избежать лишних системных вызовов для каждого отправляемого пакета.

Кроме того, если операционная система решает заменить страницу памяти в то время, как контроллер DMA копирует в нее пришедший пакет, то это приведет не только к потере пакета, но и повреждению данных в памяти.

Этих проблем можно избежать, фиксируя и освобождая страницы в памяти с помощью системных вызовов, запрещая таким образом на время их свопинг. Однако обращение к двум системным вызовам для каждого отправляемого пакета дорого. Если размер пакетов мал, например 64 байта или меньше, накладные расходы на фиксацию и открепление каждого буфера практически невозможны. Это может быть допустимо для больших пакетов, размером, скажем, 1 Кбайт или больше. Для промежуточных размеров выбор решения зависит от деталей аппаратного обеспечения [29].

В теории та же проблема возникает при работе контроллера DMA диска или другого устройства, но поскольку для них операционная система назначает свои буферы, избежать замещения страниц в системных буферах несложно. В данном случае проблема возникает, потому что пользователь настраивает контроллер DMA и управляет им, а операционная система не догадывается, что подмена страницы может оказаться фатальной. Причина, по которой использование буферов в ядре оправдано для дискового ввода-вывода, но не для коммуникаций многомашинной системы, заключается в том, что лишняя задержка в 20 мкс является приемлемой для дискового ввода-вывода, но совершенно недопустимой для связи между процессами в мультикомпьютере.

Проблемы DMA можно избежать, если пользовательский процесс сначала зафиксирует одну страницу в начале работы и спросит у системы ее физический адрес. Исходящие пакеты могут сначала копироваться туда, а затем на сетевую интерфейсную плату, но такая излишняя операция копирования почти так же нежелательна, как копирование в ядро.

По этим причинам самым безопасным методом, как правило, является использование программного ввода-вывода, так как любое страничное прерывание во время ввода-вывода представляет собой всего лишь обычное страничное прерывание центрального процессора, которое может быть обработано операционной системой стандартным образом. При возникновении страничного прерывания цикл копирования мгновенно останавливается и остается блокированным, пока операционная система не обработает страничное прерывание. Более сложная схема представляет собой использование программного ввода-вывода для небольших пакетов и прямого доступа к памяти с фиксацией и откреплением буферов для больших пакетов.

Если у сетевых карт есть свой собственный процессор (как, например, у плат фирмы Myrinet), то эти процессоры на плате могут использоваться для ускорения ввода-вывода. Однако следует избегать конфликтов между процессором на плате и центральным процессором. Один из методов избежания конфликтов показан на рис. 8.19, где узел 1 отправляет пакеты, а узел 2 получает пакеты, причем эти узлы не обязательно общаются друг с другом. Ключевой структурой синхронизации данных для отправителя является кольцо передачи, а для получателя — кольцо приема. У каждого узла есть оба кольца, так как все узлы и отправляют, и получают данные. В каждом кольце есть место для n пакетов. Кроме того, для каждого кольца поддерживается битовый массив из n битов, либо отдельно от кольца (как на рисунке), либо встроенный в кольцо. Битовый массив содержит информацию о том, которые гнезда кольца действительны в данный момент.

Когда у отправителя появляется новый пакет для отправки, он сначала проверяет, есть ли свободное гнездо в кольце. Если нет, то он должен ждать, во избежание порчи буфера. Если свободное гнездо есть, он копирует в него пакет и устанавливает соответствующий бит в битовом массиве. Закончив свою работу, процессор на плате проверяет состояние кольца передачи. Если оно содержит какие-либо пакеты, процессор выбирает из них самый старый пакет и передает его. Закончив передачу, он очищает соответствующий бит в массиве. Поскольку биты устанавливаются только центральным процессором, а очищаются только процессором на плате, конфликта не происходит. Кольцо приема работает аналогично, только теперь уже процессор на плате устанавливает бит, сообщая таким образом о при-

шедшем пакете, а центральный процессор очищает бит. Это означает, что он скопировал пакет и освободил буфер.

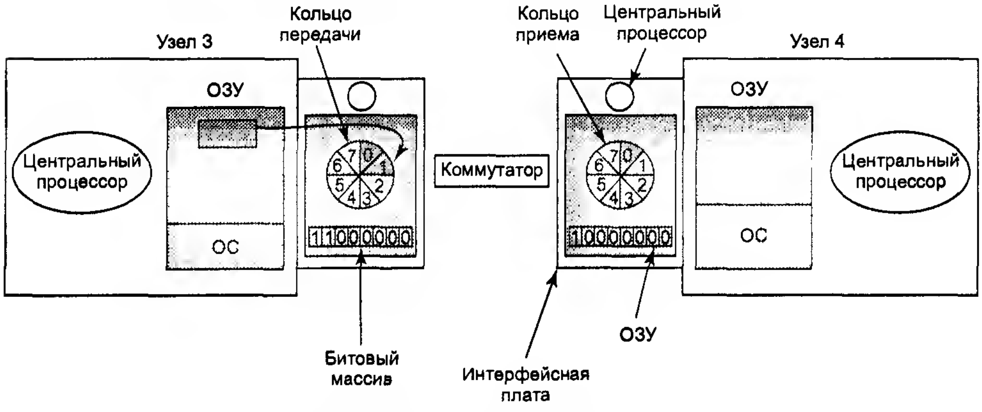


Рис. 8.19. Использование колец передачи и приема для координации центрального процессора с процессором на плате

Эта схема может также использоваться даже и без программного ввода-вывода, выполняемого центральным процессором. В этом случае кольцо передачи содержит не сам пакет, а только указатель на пакет, находящийся в оперативной памяти. Когда процессор на плате готов передавать пакет, он забирает его на интерфейсную плату либо с помощью программного ввода-вывода, либо с помощью DMA. В обоих случаях этот подход работает только тогда, когда страница, содержащая пакет, фиксирована.

Коммуникационное программное обеспечение уровня пользователя

Процессы на разных центральных процессорах многомашинной системы общаются, отправляя друг другу сообщения. В своем простейшем виде этим обменом сообщениями явно занимаются процессы пользователя. Другими словами, операционная система предоставляет способ отправки и получения сообщения, а библиотечные процедуры обеспечивают доступность этих системных вызовов для пользовательских процессов. В более сложной форме передача сообщений скрыта от пользователя под видом вызова удаленной процедуры. Ниже будут рассмотрены оба метода.

Библиотечные вызовы send и receive

Служба связи может быть минимизирована до двух (библиотечных) вызовов, один для отправки сообщений и один для их получения. Формат вызова для отправки сообщения может быть, например, таким:

```
send(dest. &mptr):
```

а вызов для получения сообщения может выглядеть так:

```
receive(addr. &mptr):
```

Первая процедура посылает сообщение, на которое указывает указатель *mptr*, процессу *dest* (идентификатор процесса) и блокирует вызывающий ее процесс до тех пор, пока сообщение не будет отправлено. Вторая процедура вызывает блокировку процесса вплоть до получения сообщения. Когда сообщение приходит, оно копируется в буфер, на который указывает *mptr*, и процесс, вызвавший эту библиотечную процедуру, разблокируется. Параметр *addr* указывает адрес, от которого вызывающий процесс ожидает прихода сообщения. Возможны различные варианты этих двух процедур и их параметров.

Вопрос состоит в том, как выполняется адресация. Так как мультикомпьютеры статичны, с фиксированным числом центральных процессоров, проще всего использовать адреса, состоящие из двух частей: номера центрального процессора и номера процесса или порта на данном центральном процессоре. Тогда каждый центральный процессор может управлять собственными адресами без конфликтов.

Блокирующие и неблокирующие вызовы

Описанные выше вызовы представляют собой **блокирующие вызовы** (иногда называемые **синхронными вызовами**). Когда процесс обращается к процедуре *send*, он указывает адресат и буфер, данные из которого следует послать указанному адресату. Пока сообщение посылается, передающий процесс блокирован (то есть приостановлен). Команда, следующая за обращением к процедуре *send*, не выполняется до тех пор, пока все сообщение не будет послано (рис. 8.20, *а*). Аналогично, обращение к процедуре *receive* не возвращает управления, пока сообщение не будет получено целиком и положено в буфер, адрес которого указан в параметре. Процесс остается приостановленным, пока не прибудет сообщение, даже если на это уйдет несколько часов. В некоторых системах получатель может указать, от кого именно он ожидает прихода сообщения. В этом случае процесс будет блокирован, пока не прибудет сообщение от указанного отправителя.

Альтернативу блокирующим вызовам составляют **неблокирующие вызовы** (иногда называемые **асинхронными вызовами**). Если процедура *send* является неблокирующей, то она возвращает управление вызывающему ее процессу практически немедленно, прежде чем сообщение будет отправлено. Преимущество этой схемы состоит в том, что отправляющий процесс может продолжать вычисления параллельно с передачей сообщения, что позволяет избежать простоя центрального процессора (при условии, что других готовых к работе процессов нет). Выбор между блокирующим и неблокирующим примитивами обычно делается проектировщиками системы (то есть, как правило, доступен либо один примитив, либо другой), хотя в некоторых системах бывают доступны оба примитива и право выбора предоставляется пользователю.

Однако помимо преимущества высокой производительности, с неблокирующими примитивами связана более сложная организация программы. Отправителю нельзя изменять содержимое буфера сообщения до тех пор, пока это сообщение не будет полностью отправлено. Последствия модификации сообщения во время его отправки слишком ужасны, чтобы рассказывать о них перед сном. Что еще хуже, если у отправляющего процесса нет возможности узнать, что передача уже выполнена, то он никогда не будет уверен, можно ли уже пользоваться буфером.

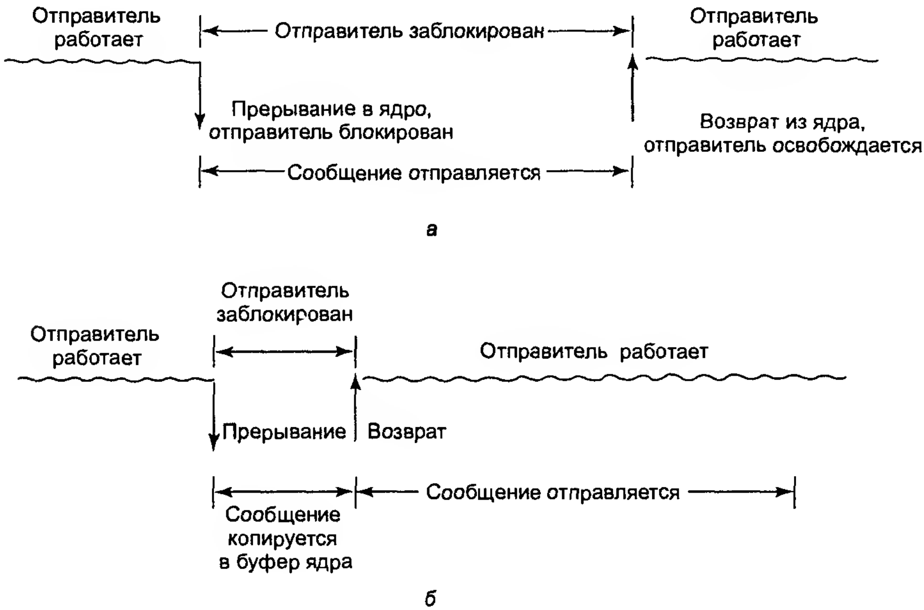


Рис. 8.20. Блокирующий вызов процедуры send (*а*); неблокирующий вызов процедуры send (*б*)

Возможны три метода решения этой проблемы. Первое решение заключается в копировании ядром сообщения в свой буфер, после чего процессу позволяется продолжать работу (рис. 8.20, *б*). С точки зрения отправителя эта схема аналогична блокирующему вызову, так как, как только отправитель получает управление, он может снова пользоваться буфером. Разумеется, сообщение еще не будет отправлено, но отправителю это не мешает. Недостаток этого метода состоит в необходимости копирования каждого исходящего сообщения из пространства пользователя в буфер ядра. Во многих сетевых интерфейсах сообщение все равно будет скопировано в аппаратный буфер передачи, поэтому первое копирование представляет собой просто потерю времени. Лишняя операция копирования может существенно снизить производительность системы.

Второе решение заключается в прерывании отправителя, когда сообщение будет отправлено, чтобы известить его об этом факте. В этом случае операции копирования не требуется, что сохраняет время, но прерывания на уровне пользователя значительно усложняют программы и могут привести к внутренним конфликтам в программе, в результате чего такие программы будет почти невозможно отладить.

Третье решение состоит в том, чтобы копировать содержимое буфера при записи. Буфер помечается как доступный только для чтения до тех пор, пока сообщение не будет отправлено. Если буфер используется повторно прежде, чем будет отправлено сообщение, создается копия буфера. Недостаток этого варианта заключается в том, что если только буферу не выделена целиком собственная страница, операции записи с соседними переменными также будут вызывать копирование страницы. Кроме того, потребуются дополнительные административные меры,

так как теперь отправка сообщения неявно изменяет статус чтения/записи страницы. Наконец, раньше или позже, страница опять может быть записана, что приведет к появлению еще одной копии страницы.

Таким образом, у отправителя имеется следующий выбор:

1. Блокирующая операция *send* (центральный процессор простаивает во время передачи сообщения).
2. Неблокирующая операция *send* с копированием (время центрального процессора теряется на создание дополнительной копии).
3. Неблокирующая операция *send* с прерыванием (усложняет программу).
4. Копирование при записи (в конечном итоге требуется дополнительная операция копирования).

При нормальных условиях первый вариант является лучшим, особенно при наличии нескольких потоков. При этом пока один поток блокирован, ожидая отправки сообщения, остальные потоки могут продолжать работу. Для этого метода также не требуется буфера в ядре. Более того, как можно увидеть, сравнивая рис. 8.20, *а* и рис. 8.20, *б*, передача сообщения занимает меньше времени, если не требуется дополнительного копирования.

Следует отметить, что различными авторами используются различные критерии для того, чтобы отличать синхронные примитивы от асинхронных. Альтернативная точка зрения заключается в том, что вызов является синхронным, только если отправитель блокируется до тех пор, пока не будет отправлено сообщение и не будет получено подтверждение [13]. В сфере коммуникаций реального времени синхронность имеет другое значение, что, к сожалению, приводит к путанице.

Как и *send*, операция *receive* также может быть блокирующей и неблокирующей. Блокирующий вызов просто приостанавливает процесс до прибытия сообщения. Если на центральном процессоре одновременно работают несколько потоков, то такой подход является наиболее простым. Неблокирующий вызов *receive* лишь сообщает ядру, где расположен буфер, и почти сразу же возвращает управление. При прибытии сообщения для извещения процесса может использоваться прерывание. Однако прерывания сложно программировать, к тому же они довольно медленны. Поэтому, возможно, лучшим решением является периодическое обращение к почтовому ящику при помощи процедуры *poll* (опрос), сообщающей, пришли ли какие-либо сообщения. Если есть новые сообщения, получатель может забрать их с помощью процедуры *get_message* (получить сообщение), возвращающей первое прибывшее сообщение. В некоторых системах компилятор может сам вставлять вызовы процедуры *poll* в соответствующих местах программы, хотя определить, как часто следует обращаться к этой процедуре, непросто.

Еще один вариант заключается в том, что при прибытии сообщения в адресном пространстве получающего процесса создается новый поток. Такой поток называется **всплывающим** или **временным потоком**. Он выполняет заранее указанную процедуру, в качестве параметра которой передается указатель на прибывшее сообщение. После обработки сообщения этот процесс просто прекращает свое существование.

Вариантом этой идеи является запуск программы получателя прямо в обработчике прерываний, что позволяет избежать хлопот с созданием временного потока. Чтобы еще ускорить эту схему, в само сообщение можно включить адрес обработчика, поэтому, когда оно прибудет, обработчик будет вызван с помощью всего нескольких команд процессора. Большой выигрыш данной схемы заключается в том, что копирование вообще не нужно. Обработчик получает сообщение от интерфейсной платы и обрабатывает его на лету. Такая схема называется **активными сообщениями** [348]. Поскольку каждое сообщение содержит адрес обработчика, такая схема может работать только в том случае, когда отправители и получатели полностью доверяют друг другу.

Вызов удаленной процедуры

Хотя модель передачи сообщений предоставляет удобный способ структурирования операционной системы многомашинной системы, у нее есть один существенный недостаток: связь между процессами построена на парадигме ввода-вывода. Процедуры *send* и *receive* занимаются вводом-выводом, а многие программисты считают ввод-вывод неверной моделью программирования.

Эта проблема известна уже давно, но мало что делалось в этом направлении вплоть до выхода статьи Биррелла и Нельсона [33], в которой был предложен принципиально другой способ решения данной проблемы. Хотя сама идея довольно проста (после того, как кто-то другой уже нашел решение), последствия ее реализации иногда не совсем очевидны. В данном разделе мы рассмотрим концепцию, ее реализацию, обсудим ее достоинства и недостатки.

В двух словах, то, что предложили Биррелл и Нельсон, заключалось в разрешении программам вызывать процедуры, расположенные на других машинах. Когда процесс на машине 1 вызывает процедуру на машине 2, вызывающий процесс на машине 1 приостанавливается, а на машине 2 выполняется вызванная процедура. Информация между вызывающим процессом и вызываемой процедурой может передаваться через параметры, а также возвращаться в результате процедуры. Программисту не видны ни передача сообщений, ни операции ввода-вывода. Такая техника, называемая **вызовом удаленной процедуры** (RPC, Remote Procedure Call), стала основой большого количества программного обеспечения для многомашинных систем. Традиционно вызывающую процедуру называют клиентом, а вызываемую — сервером. Мы также будем использовать эту терминологию.

Идея удаленного вызова процедуры заключается в том, что вызов удаленной процедуры должен выглядеть максимально похоже на вызов локальной процедуры. В простейшей форме для вызова удаленной процедуры клиентская программа должна быть связана с небольшой библиотечной процедурой, называемой **клиентским суррогатом** или **клиентской заглушкой**, представляющей серверную процедуру в адресном пространстве клиента. Аналогично, сервер связан с процедурой, называемой **серверным суррогатом** или **серверной заглушкой**. Эти процедуры создают видимость локальности вызова.

Фактические этапы выполнения вызова удаленной процедуры показаны на рис. 8.21. Суррогаты на рисунке затенены серым цветом. Шаг *1* состоит в обраще-

нии клиента к клиентскому суррогату. При этом параметры передаются через стек, как при обычном вызове. На шаге *2* клиентский суррогат упаковывает параметры в сообщение и обращается к системному вызову для оправки сообщения. Упаковка параметров называется **маршалингом** или **маршализацией**. На шаге *3* ядро посылает сообщение с клиентской машины на сервер. На шаге *4* ядро серверной машины передает полученное сообщение серверному суррогату (который уже ожидает прихода сообщения). Наконец, на шаге *5* серверный суррогат вызывает серверную процедуру. Ответ проходит по тому же пути в обратном направлении.

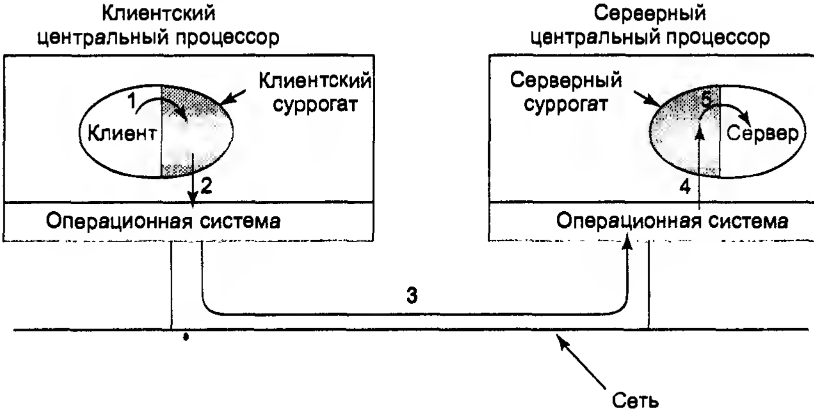


Рис. 8.21. Этапы выполнения вызова удаленной процедуры

Ключевой вопрос, на который следует здесь обратить внимание, состоит в том, что клиентская процедура, написанная пользователем, выполняет нормальный (то есть локальный) процедурный вызов клиентского суррогата. Так как клиентская процедура и клиентский суррогат находятся в одном адресном пространстве, параметры передаются обычным образом. Аналогично, серверная процедура вызывается процедурой в своем адресном пространстве. Таким образом, вместо выполнения ввода-вывода с помощью процедур *send* и *receive*, связь с удаленными объектами осуществляется при помощи имитации нормальных процедурных вызовов.

Вопросы реализации

Несмотря на концептуальную элегантность вызова удаленной процедуры, данный метод содержит множество незаметных на первый взгляд проблем. Одна из них заключается в использовании указателей в качестве параметров. В нормальных условиях передача указателя в качестве параметра не представляет проблемы. Вызываемая процедура может использовать параметр тем же способом, что и процесс, вызывающий процедуру, поскольку обе процедуры находятся в одном виртуальном адресном пространстве. При использовании вызова удаленной процедуры передача указателя невозможна, так как клиент и сервер находятся в различных адресных пространствах.

В некоторых случаях указатели могут передаваться при помощи специальных ухищрений. Предположим, что первый параметр представляет собой указатель на

целое число k. Клиентский суррогат может упаковать само число k в сообщение и передать его серверу. Затем серверный суррогат создает указатель на число k и передает его серверной процедуре, как и ожидалось. Когда серверная процедура возвращает управление серверному суррогату, тот оправляет число k назад клиенту, где новое число k копируется поверх старого. В результате стандартная передача параметров по ссылке была заменена передачей значений. К сожалению, этот метод не всегда работает, например не работает в том случае, когда указатель указывает на какую-либо сложную структуру. По этой причине на параметры удаленных процедур должны быть наложены определенные ограничения.

Вторая проблема заключается в том, что в языках со слабым контролем типов, таких как С, вполне допустимо написание процедуры, вычисляющей скалярное произведение двух векторов (массивов), без указания размеров векторов. Массивы могут, например, ограничиваться специальным символом, известным только вызывающей и вызываемой процедурам. При такой ситуации клиентский суррогат не способен корректно упаковать в сообщение все необходимые параметры, так как у него нет способа определить их размеры.

Третья проблема состоит в том, что типы параметров не всегда могут быть точно определены, даже из формального описания самой программы. Так, например, процедура *printf* может иметь произвольное количество параметров (по меньшей мере, один), которые могут быть произвольной смесью целых и вещественных чисел, переменных и констант типа short, long, символьных и строковых переменных и констант произвольной длины, а также других типов. Вызов *printf* как удаленной процедуры практически невозможен, поскольку в языке С разрешено столь многое. Однако правило, говорящее, что вызов удаленных процедур может быть использован при условии, что вы не программируете на С (или С++), вряд ли найдет понимание среди широких масс программистов.

Четвертая проблема относится к использованию глобальных переменных. При нормальных условиях вызывающая и вызываемая процедуры могут общаться, используя, помимо передаваемых параметров, глобальные переменные. Если теперь вызываемая процедура перемещается на другую машину, программа не сможет работать, так как глобальные переменные окажутся на разных машинах.

Наличие всех этих проблем не означает, что дело вызова удаленных процедур безнадежно. В действительности этот метод широко применяется, но для его корректной работы необходимо соблюдение определенных ограничений.

Распределенная память совместного доступа

Несмотря на привлекательность вызова удаленных процедур, многие программисты все еще предпочитают модель совместно используемой памяти и хотели бы применять ее даже на многомашинной системе. Это может показаться удивительным, но даже при отсутствии общей памяти можно сохранить иллюзию ее наличия при помощи техники, называемой DSM (Distributed Shared Memory — распределенная совместно используемая память) [204, 205]. В DSM каждая страница находится в одном из блоков памяти (см. рис. 8.1). У каждой машины есть своя виртуальная память и собственные таблицы страниц. Когда центральный процессор выполняет команды LOAD и SAVE в странице, которой у него нет, происходит странич-

ное прерывание с передачей управления операционной системе. Операционная система находит страницу и просит центральный процессор, владеющей ею на данный момент, выгрузить страницу и переслать ее по соединительной сети. Когда страница прибывает, она загружается в память, а команда, вызвавшая страничное прерывание, перезапускается. В результате операционная система просто удовлетворяет страничное прерывание, но не с локального диска, а по сети с удаленного блока памяти. С точки зрения пользователя все это выглядит так, как если бы у машины была совместно используемая память.

Различие между настоящей общей памятью и системой DSM показано на рис. 8.22. На рис. 8.22, *а* изображена настоящая многомашинная система с аппаратно реализованной общей памятью. На рис. 8.22, *б* показана система DSM, реализуемая операционной системой. Рисунок 8.22, *в* демонстрирует еще одну форму совместно используемой памяти, на этот раз реализованную более высокими уровнями программного обеспечения. Такой вариант будет рассматриваться позднее в этой главе, но на данный момент мы сконцентрируемся на системе DSM.

Обсудим теперь некоторые подробности работы системы DSM. В DSM адресное пространство разделяется на страницы, которые распределены между всеми узлами системы. Когда центральный процессор обращается к адресу, не являющемуся локальным, происходит прерывание и программное обеспечение DSM добывает страницу, содержащую данный адрес, и перезапускает команду, вызвавшую страничное прерывание, которая во второй раз завершается успешно. Эта концепция для адресного пространства, состоящего из 16 страниц и четырех узлов, каждый из которых способен удерживать пять страниц, проиллюстрирована на рис. 8.23, *а*.

В данном примере, если центральный процессор 0 обращается к командам или данным на страницах 0, 2, 5 или 9, эти обращения выполняются локально. Обращения к другим страницам вызывают прерывания. Например, обращение по адресу в странице 10 вызовет прерывание с передачей управления программному обеспечению DSM, которое перемещает страницу 10 от узла 1 узлу 0 (рис. 8.23, *б*).

Репликация

Усовершенствование, способное значительно повысить производительность системы, состоит в репликации страниц, для которых разрешено только чтение, например с текстом программы или константами. Если страница 10 на рис. 8.23 представляет собой программную секцию, то при обращении к ней центрального процессора 0 может быть создана копия, посылаемая на процессор 0, так что память центрального процессора 1 не изменяется (рис. 8.23, *в*). Таким образом, центральные процессоры 0 и 1 могут оба обращаться к странице 10 настолько часто, насколько это им необходимо, не вызывая в дальнейшем страничных прерываний.

Другая возможность заключается в том, чтобы реплицировать все страницы, а не только страницы с доступом только для чтения. Пока эти страницы только читаются, нет никакой разницы между репликацией страниц, которые можно модифицировать, и страниц, которые модифицировать нельзя. Но как только реплицированная страница модифицируется, следует предпринять специальные действия во избежание появления двух различных копий одной страницы. Поддержка непротиворечивости системы будет обсуждаться в следующих разделах.



Рис. 8.22. Различные уровни, на которых может быть реализована совместно используемая память: аппаратный (*а*); операционная система (*б*); программное обеспечение уровня пользователя (*в*)

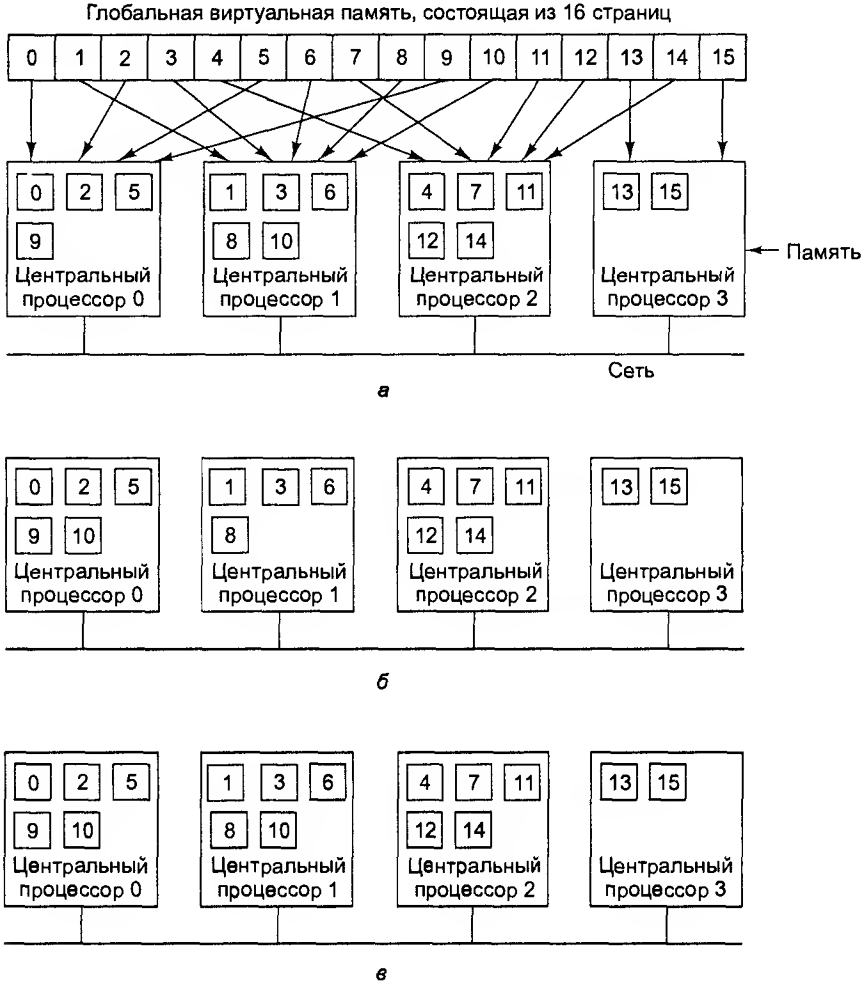


Рис. 8.23. Страницы адресного пространства, распределенные между машинами (*а*); центральный процессор 0 обратился к странице 10 (*б*); если страница 10 доступна только для чтения, то используется репликация (а)

Ложное совместное использование памяти

Системы DSM во многом напоминают многомашинные системы. В обеих системах при обращении к слову нелокальной памяти блок памяти, содержащий это слово, берется с текущего местоположения и помещается на машину, с которой было произведено обращение. Важный вопрос заключается в том, насколько большим должен быть этот блок памяти? В многомашинных системах размер блока кэша, как правило, составляет 32 или 64 байта, чтобы не оказывать слишком большой нагрузки на шину. В системах DSM размер передаваемого по сети блока памяти должен быть кратен размеру страницы (так как диспетчер памяти работает со страницами). Подобная работа имитирует страницы большего размера.

У большего размера страниц в системе DSM есть как преимущества, так и недостатки. Основное преимущество заключается в том, что время, требуемое для переноса по сети 4096 байт, не намного больше, чем для переноса 1024 байт, так как существенное время тратится на подготовку этого процесса. Увеличивая передаваемые по сети блоки, часто можно уменьшить количество передаваемых блоков. Это свойство особенно важно, так как многие программы обладают локальностью ссылок, а это означает, что если программа обратилась к какому-нибудь слову на странице, то велика вероятность, что в ближайшем будущем она обратится также и к другим словам на этой же странице.

С другой стороны, блоки памяти больших размеров будут дольше передаваться по сети, блокируя страничные прерывания других процессов. Кроме того, слишком большой размер логических страниц вызывает новую проблему, называемую **ложным совместным использованием** (рис. 8.24). Здесь на одной странице содержатся две не имеющие друг к другу отношения переменные, *A* и *B*. Процессор 1 часто пользуется переменной *A*, читая и записывая ее. А процессор 2 часто использует переменную *B*. В такой ситуации страница, содержащая обе переменные, будет постоянно путешествовать от одной машины к другой.

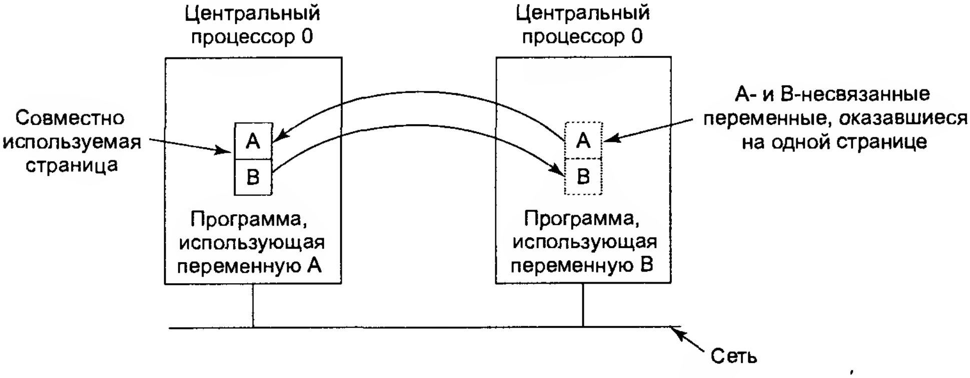


Рис. 8.24. Ложное совместное использование страницы, содержащей две несвязанные переменные

Данная проблема заключается в том, что хотя эти переменные и не связаны друг с другом, они оказались на одной странице, поэтому когда какой-либо процесс использует одну из этих переменных, он вместе с ней получает и другую. Чем больше эффективный размер страницы, тем выше вероятность ложного совместного использования памяти, и наоборот — чем меньше эффективный размер страницы, тем реже будет возникать подобная ситуация. В обычной системе виртуальной памяти нет ничего подобного данному феномену.

Умные компиляторы, понимающие суть данной проблемы, стараются помещать переменные разных процессов в различные страницы, что помогает снизить частоту ситуаций ложного совместного использования памяти и увеличить производительность. Однако сказать это легче, чем сделать. Более того, если ситуация ложного совместного использования заключается в том, что узел 1 использует один элемент массива, а узел 2 — другой элемент того же массива, то даже умный компилятор не сможет устранить эту проблему.

Последовательная непротиворечивость

Если модифицируемые страницы не реплицируются, поддержание непротиворечивости системы не представляет проблем. У каждой модифицируемой страницы есть ровно одна копия, которая динамически перемещается по мере надобности от одного узла к другому. Поскольку не всегда есть возможность заранее предсказать, какая страница будет модифицироваться, во многих системах DSM при попытке процессом прочитать удаленную страницу создается локальная копия. При этом обе копии отмечаются в своих диспетчерах памяти как доступные только для чтения. Пока все процессы обращаются к этим страницам только для чтения, никаких проблем не возникает.

Однако как только какой-либо процесс пытается записать реплицированную страницу, возникает потенциальная проблема непротиворечивости, потому что изменение только одной копии неприемлемо. Эта ситуация подобна тому, что происходит в многомашинной системе, когда один центральный процессор пытается модифицировать слово, присутствующее в нескольких кэшах. В данном случае решение заключается в том, что центральный процессор, собирающийся выполнить запись, сначала подает на шину сигнал, сообщающий всем остальным центральным процессорам, что их копия неверна и ее нужно удалить из блока кэша. Система DSM работает практически так же. Прежде чем общая страница может быть записана, всем остальным центральным процессорам, владеющим копией этой страницы, посылается сообщение, в котором предлагается выгрузить и аннулировать имеющуюся у них страницу. Когда все центральные процессоры ответят на это сообщение, оригинальный центральный процессор может выполнить операцию записи.

Другой способ поддержать наличие нескольких копий модифицируемых страниц заключается в использовании мьютексов. Чтобы получить возможность писать в какую-либо часть виртуального пространства, процесс должен сначала получить мьютекс. После этого процесс может читать и писать в эту память столько раз, сколько это ему необходимо. Когда блокировка отпускается, изменения распространяются на другие копии. До тех пор пока только один центральный процессор может получить блокировку для записи в страницу, такая схема обеспечивает непротиворечивость данных.

Альтернативный метод состоит в том, что при первой записи в страницу создается ее копия, которая сохраняется на центральном процессоре, выполняющем запись. Когда другой центральный процессор на удаленной машине пытается получить блокировку к этой странице, процессор, писавший в эту страницу раньше, сравнивает текущее состояние этой страницы с чистым оригиналом и отсылает этому центральному процессору список изменений страницы. Таким образом, второй процессор вместо аннулирования страницы может обновить ее [178].

Планирование многомашинных систем

На мультипроцессоре все процессы располагаются в общей памяти. Когда какой-либо центральный процессор завершает текущее задание, он выбирает процесс и запускает его. В принципе все процессы являются потенциальными кандидатами. В многомашинной системе ситуация в корне отличается. У каждого узла есть

своя собственная память и свой собственный набор процессов. Центральный процессор 1 не может внезапно решить запустить процесс, расположенный на узле 4, не приложив для этого достаточно усилий. Это отличие означает, что планирование на многомашинной системе реализуется проще, но распределение процессов между узлами представляет собой более важную задачу. Ниже мы обсудим эти вопросы.

Планирование многомашинной системы похоже на планирование мультипроцессора, но не все рассматривавшиеся ранее алгоритмы планирования одной системы применимы к другой системе. Однако простейший мультипроцессорный алгоритм — учет всех готовых процессов в едином централизованном списке — не будет работать в многомашинной системе, так как каждый процесс может работать только на том центральном процессоре, на котором он расположен в данный момент. Тем не менее при создании нового процесса можно выбрать центральный процессор, на котором он будет работать, например, чтобы сбалансировать нагрузку.

Поскольку у каждого узла есть свои процессы, можно использовать любой локальный алгоритм планирования. Однако возможно применение бригадного планирования так же, как оно используется на мультипроцессоре, потому что для него требуется всего лишь изначальное принятие решения о том, какой процесс в котором интервале времени будет работать, и некий способ синхронизации начала всех временных интервалов.

Балансировка нагрузки

О планировании многомашинных систем можно сказать не так уж и много, потому что как только процесс назначается какому-либо узлу, может использоваться любой локальный алгоритм планирования. Однако именно потому, что так мало можно сделать после того, как процесс уже назначен узлу, решение о выборе узла представляет такую важность. В этом основное отличие многомашинных систем от мультипроцессоров, в которых все процессы работают в одной памяти и могут переключаться на любой центральный процессор уже во время исполнения. Соответственно, следует определить, как назначать процессы узлам, чтобы при этом повысить эффективность системы. Алгоритмы и эвристики для назначения процессов узлам называются **алгоритмами распределения процессоров**.

За несколько лет было разработано большое количество алгоритмов распределения процессоров (то есть узлов). Различаются они тем, что предполагается известным в том или ином алгоритме и чего требуется достичь. К известным свойствам процессов относятся процессорные требования, количество памяти и объем обмена информацией с другими процессами. Возможные задачи включают минимизацию потерянного процессорного времени, вызванного отсутствием локальной работы, минимизацию суммарного сетевого трафика, а также гарантирование справедливого распределения ресурсов для пользователей и процессов. Ниже будут рассмотрены некоторые алгоритмы, позволяющие получить представление об этих возможностях.

Детерминистический графовый алгоритм

Хорошо изучен класс алгоритмов для систем, состоящих из процессов с известными требованиями к центральному процессору и памяти, а также матрицей известных объемов обмена данными между каждой парой процессов. Если количество процессов больше количества центральных процессоров, то некоторые процессы должны быть назначены каждому процессору. Идея заключается в том, чтобы минимизировать сетевой трафик.

Система может быть представлена в виде взвешенного графа, каждая вершина которого представляет собой процесс, а каждая дуга — поток сообщений между двумя процессами. Математически проблема сводится к тому, чтобы найти способ разбиения графа на *k* непересекающихся подграфов при определенных ограничениях, накладываемых на подграфы (например, суммарные требования центральных процессоров и памяти для подграфа не должны превышать некоторого установленного предела). Для каждого решения, удовлетворяющего требованиям, дуги, находящиеся целиком внутри подграфа, представляют внутримашинный обмен информацией и могут игнорироваться. Дуги, идущие от одного подграфа к другому, представляют сетевой трафик. Цель состоит в том, чтобы найти такой вариант разбиения графа на подграфы, который минимизирует сетевой трафик при выполнении всех требований. Так, на рис. 8.25 показана система из девяти процессов от *A* до *I*. На дугах отмечены значения сетевой нагрузки между процессами (например, в мегабитах в секунду).

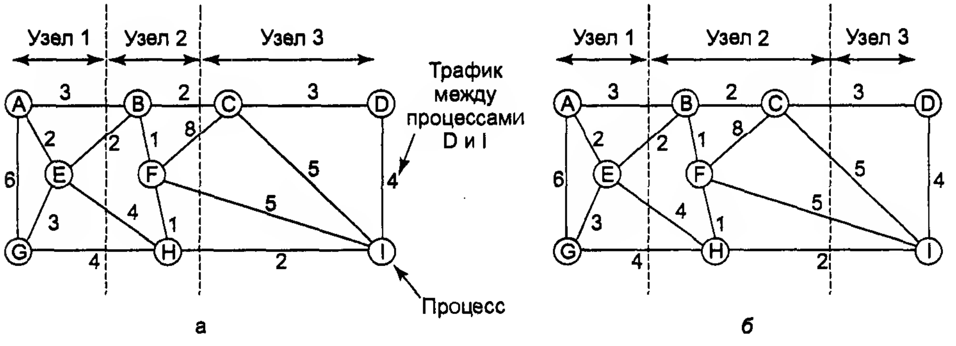


Рис. 8.25. Два способа распределения девяти процессов на трех узлах

На рис. 8.25, *а* граф разделен следующим образом: узлу 1 назначены процессы *A*, *E* и *G*, на узле 2 работают процессы *B*, *F* и *H*, а узлу 3 достались процессы *C*, *D* и *I*. Общий сетевой трафик представляет собой сумму весов дуг, пересеченных границами подграфов (показаны штриховыми линиями на рисунке). В данном случае он равен 30. На рис. 8.25, *б* представлен другой вариант разбиения графа на подграфы. При условии, что такой вариант распределения процессов удовлетворяет всем требованиям памяти и процессорной мощности, этот выбор лучше, так как при нем требуется меньший сетевой трафик.

Для оптимального разбиения графа на отдельные части следует, видимо, искать группы тесно связанных процессов (с высоким уровнем внутригруппового трафика), но мало взаимодействующие с другими группами (низкий межгрупповой трафик). Данная проблема обсуждается в следующих статьях: [68, 213, 318].

Распределенный эвристический алгоритм, инициируемый отправителем

Рассмотрим теперь некоторые распределенные алгоритмы. При использовании одного алгоритма процесс работает на узле, который его создал, если только этот узел не перегружен. Перегрузка может оцениваться по числу процессов, их суммарной нагрузке на процессор или по какому-либо еще параметру. Если узел оказывается перегружен, он выбирает случайным образом другой узел и запрашивает у него данные о его загрузке. Если загрузка данного узла оказывается ниже определенного уровня, новый процесс отправляется туда [102]. Если данный узел также уже достаточно загружен, выбирается другая машина. Смысл в том, чтобы перегруженные узлы могли попытаться избавится от лишней нагрузки (рис. 8.26, *а*).

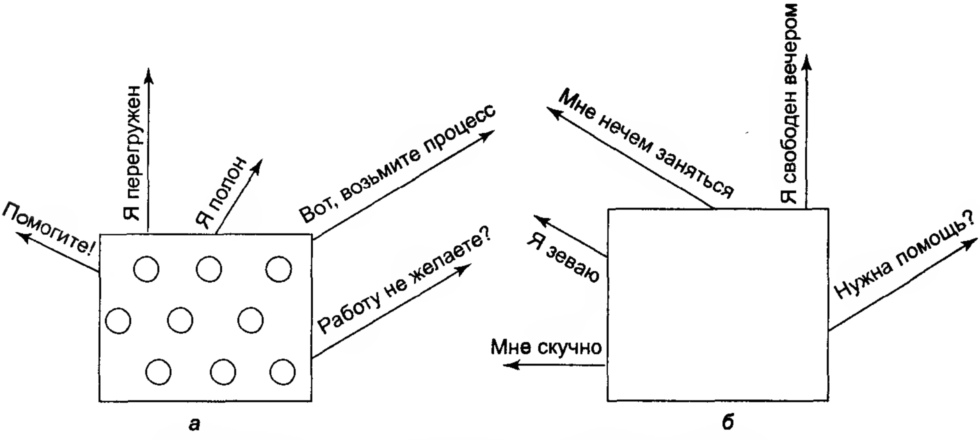


Рис. 8.26. Перегруженный узел ищет легко загруженный узел, которому можно передать процесс (*а*); пустой узел ищет работу (*б*)

Авторы статьи, на которую мы ссылались выше, создали для данного алгоритма аналитическую модель очередей. С помощью этой модели было установлено, что алгоритм хорошо работает и обладает устойчивостью в широком диапазоне параметров, к которым относятся различные пороговые величины и значения стоимости переноса данных.

Тем не менее следует отметить, что в условиях сильной загруженности все машины будут постоянно посылать другим машинам запросы об их загрузке в тщетной попытке найти кого-нибудь, кто согласится взять у них часть работы. Мало кому из процессов удастся уменьшить свою нагрузку, но сами попытки поиска помощников могут оказать существенную дополнительную нагрузку на сеть.

Распределенный эвристический алгоритм, инициируемый получателем

Существует алгоритм, схожий с вышеописанным, но в котором передача процесса менее загруженной машине инициируется получателем, то есть менее загруженной машиной (см. рис. 8.26, *а*). Когда завершается очередной процесс, при таком алго-

ритме система проверяет, достаточно ли у нее работы. Если нет, она случайным образом выбирает машину и просит у нее работу. Если этой машине нечего предложить, проверяется другая машина, затем третья и т. д. Если за N попыток работу найти не удалось, узел временно прекращает поиск, выполняет любую работу, которая у него есть, после чего весь цикл повторяется после завершения очередного процесса. Если работы у узла не остается совсем, он простаивает некоторое время, а спустя определенный интервал времени снова принимается искать работу.

Преимущество этого алгоритма заключается в том, что он не оказывает дополнительной нагрузки на сеть в критический период. Предыдущий алгоритм обращался с новыми запросами в сеть как раз тогда, когда загрузка узлов и сети была и без того высока. В данной схеме вероятность появления незагруженной машины в перегруженной системе мала. Однако если такое вдруг случается, простаивающая машина легко найдет себе работу в такой ситуации. Конечно, когда почти вся система простаивает, алгоритм, в котором инициатором выступает получающая сторона, создает существенный трафик поиска работы. И все же значительно лучше иметь накладные расходы в недогруженной системе, чем в перегруженной.

Возможна комбинация обоих алгоритмов, в которой машины будут пытаться избавиться от лишней работы и получить работу, когда ее не хватает. Более того, машины, вероятно, могут улучшить эффективность случайного опроса, храня историю последних попыток и определяя машины, страдающие от хронической перегрузки или, наоборот, недогруженности. В зависимости от того, хочет ли инициатор запроса получить работу или избавиться от нее, он может попытаться связаться в первую очередь с одной из таких машин.

Алгоритм торгов

Другой класс алгоритмов пытается превратить компьютерную систему в некое подобие миниатюрной экономики, с продавцами и покупателями услуг и ценами, устанавливаемыми спросом и предложением [115]. Ключевую роль в экономике играют процессы, которые должны покупать процессорное время для выполнения своей работы, и узлы, продающие свои циклы процессоров на аукционе тому, кто предложит наивысшую цену.

Каждый узел рекламирует свою приблизительную цену, публикуя ее в файле, доступном для чтения всем процессам. Эта цена не является фиксированной, но она дает представление об уровне предоставляемых услуг (в действительности это цена, которую заплатил предыдущий покупатель). У разных узлов цены могут различаться в зависимости от их быстродействия, объема оперативной памяти, наличия быстрого аппаратного обеспечения для обработки операций с плавающей точкой и других свойств. Могут публиковаться и такие характеристики, как ожидаемое время отклика.

Когда процесс хочет запустить дочерний процесс, он ищет узел, предлагающий требующиеся ему в данный момент услуги. Затем он определяет набор узлов, услугами которых он может воспользоваться. Из этого набора процесс выбирает лучшего кандидата, где слово «лучший» может означать самого дешевого, быстро-

го или с наилучшим соотношением производительность/цена. Процесс формирует заявку с предложением цены и посылает ее первому кандидату. Предлагаемая цена может быть выше или ниже объявленной цены.

Процессоры собирают все заявки, посланные им, и делают свой выбор, как правило, останавливаясь на заявке с самой высокой ценой. Победители и проигравшие информируются о сделанном выборе. Победивший процесс исполняется. Затем обновляется опубликованная цена сервера.

Хотя авторы указанной выше статьи не вдаются в подробности, подобная экономическая модель поднимает ряд интересных вопросов, например: где процессы берут деньги на покупку услуг? Регулярно ли им выплачивают зарплату? Получают ли они одинаковые деньги, или деканы получают больше профессоров, а студенты, как всегда, получают меньше всех? Если в систему поступают новые пользователи, а количество ресурсов не увеличивается, вызывает ли это рост цен (инфляцию)? Могут ли узлы объединяться в картели для вздутия цен? Разрешены ли объединения пользователей в союзы? Вводится ли плата за дисковое пространство и печать на принтере? Стоит ли печать изображений больше, чем печать текста? Этот список вопросов можно продолжить.

Распределенные системы

Мы завершили наше знакомство с мультипроцессорами и многомашинными системами. Теперь настала пора обсудить третий тип многопроцессорных систем: **распределенные системы**. Эта система сходна с многомашинной системой в том, что в такой системе нет общей физической памяти, а у каждого узла есть своя память. Но в отличие от многомашинной системы, узлы распределенной системы связаны друг с другом не столь жестко.

Во-первых, узел многомашинной системы, как правило, содержит центральный процессор, оперативную память, сетевую интерфейсную плату и, возможно, жесткий диск для выгрузки страниц памяти. В отличие от него, узел распределенной системы представляет собой полноценный компьютер, с полным набором периферийных устройств. Во-вторых, узлы многомашинной системы обычно располагаются в одном помещении, что позволяет соединить их высокоскоростной сетью, тогда как узлы распределенной системы могут быть распределены по всему миру. Наконец, все узлы многомашинной системы работают под управлением одной операционной системы, совместно используют единую файловую систему и находятся под общим административным управлением. На узлах же распределенной системы могут работать различные операционные системы, у каждого узла своя файловая система, и администрация различных компьютеров также может быть разной. Типичный пример многомашинной системы — это 512 узлов в одной комнате в компании или университете, занимающихся, скажем, фармацевтическим моделированием, тогда как типичный пример распределенной системы состоит из тысяч машин, общающихся по Интернету. В табл. 8.1 сравниваются мультипроцессоры, многомашинные системы и распределенные системы.

Таблица 8.1. Сравнение трех типов многопроцессорных систем

Аспект	Мультипроцессор	Многомашинная система	Распределенная система
Конфигурация узла	Центральный процессор	Центральный процессор, ОЗУ, сетевой интерфейс	Полный компьютер
Периферия узла	Все общее	Общая, кроме, может быть, дисков	Полный набор для каждого узла
Расположение	В одном блоке	В одном помещении	Возможно, по всему миру
Связь между узлами	Общая память	Выделенные линии	Традиционная сеть
Операционные системы	Одна, общая	Несколько, одинаковые	Могут быть различными
Файловые системы	Одна, общая	Одна, общая	У каждого узла своя
Администрирование	Одна организация	Одна организация	Много организаций

Многомашинные системы занимают в данной таблице середину. Возникает интересный вопрос: «К чему ближе многомашинные системы, к мультипроцессорам или к распределенным системам?» Как ни странно это может показаться, но ответ зависит от вашей точки зрения. С технической точки зрения у мультипроцессоров есть общая память, которой нет у остальных двух видов многопроцессорных систем. Результатом этого отличия является использование различных программных моделей. Однако с прикладной точки зрения мультипроцессоры и многомашинные системы представляют собой просто несколько больших шкафов с аппаратурой, расположенных в одном помещении. И та и другая система используются для решения задач, требующих больших объемов вычислений, тогда как распределенная система, соединяющая компьютеры по Интернету, в большей степени занимается связью, чем вычислениями, и используется иначе.

Слабое соединение компьютеров в распределенной системе одновременно является и их силой, и узким местом. Сильная сторона заключается в том, что компьютеры могут использоваться для широкого спектра приложений. Слабость состоит в том, что программирование таких приложений затрудняется отсутствием лежащей в основе общей модели.

К типичным Интернет-приложениям относятся доступ к удаленному компьютеру (при помощи программ *telnet* и *rlogin*), доступ к удаленной информации (с помощью Всемирной паутины и протокола FTP), средства связи (электронная почта и чат) и многочисленные новые приложения (электронная коммерция, телемедицина и дистанционное обучение). Недостаток всех этих приложений состоит в том, что для каждого из них приходится изобретать колесо. Например, электронная почта, FTP и Всемирная паутина в основном занимаются перемещением файлов из пункта *A* в пункт *B*, но каждое из этих приложений делает это по-своему, используя свои соглашения об именах, свои протоколы передачи, технику копирования и т. д. Хотя многие web-браузеры скрывают эти различия от пользователей, лежащие в основе приложений механизмы совершенно различны. Сокрытие различий на уровне интерфейса пользователя подобно тому, как если бы кто-либо купил билет из Нью-Йорка в Сан-Франциско и только позднее обнаружил бы, был ли это билет на самолет, поезд или автобус.

Распределенная система добавляет к лежащей в основе сети некую общую парадигму (модель), обеспечивающую однородный вид всей системы. Цель распределенной системы состоит в том, чтобы превратить множество слабосвязанных машин в когерентную систему, основанную на единой концепции. Иногда парадигма является простой, а иногда более сложной, но идея всегда заключается в том, чтобы предоставить нечто, объединяющее систему.

Простой пример объединяющей парадигмы в слегка ином контексте можно обнаружить в системе UNIX, в которой все устройства ввода-вывода выглядят как файлы. Возможность управлять всеми клавиатурами, принтерами и линиями последовательной передачи одним и тем же способом упрощает взаимодействие с этими устройствами.

Один из способов, с помощью которого распределенная система может достичь определенного уровня однородности, несмотря на разницу в лежащем в основе аппаратном обеспечении, заключается в установке специального уровня программного обеспечения поверх операционной системы. Этот уровень, называемый **промежуточным программным обеспечением**, а также **связующим** или **посредническим программным обеспечением** (middleware), проиллюстрирован на рис. 8.27. Уровень предоставляет определенные структуры данных и операции, позволяющие процессам и пользователям на сильно удаленных машинах взаимодействовать друг с другом.

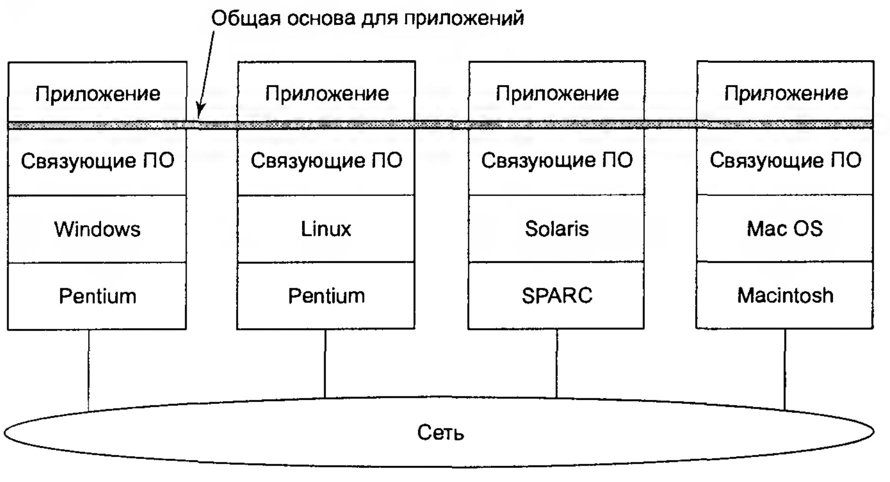


Рис. 8.27. Положение промежуточного программного обеспечения в распределенной системе

В определенном смысле промежуточное программное обеспечение подобно операционной системе распределенной системы. С другой стороны, оно *не является* операционной системой, поэтому мы не станем углубляться в детали в данной книге. Подробнее о распределенных системах можно прочитать в [325]. В оставшейся части этой главы мы кратко познакомимся с аппаратным обеспечением распределенных систем (то есть с лежащей в их основе компьютерной сетью), а затем с программным обеспечением связи (сетевыми протоколами). После этого мы рассмотрим несколько парадигм, используемых в данных системах.

Сетевое аппаратное обеспечение

Распределенные системы создаются поверх компьютерных сетей, поэтому следует привести краткое введение в данный предмет. Существует два основных типа сетей: **локальные сети** (LAN, Local Area Network), покрывающие одно здание или территорию учреждения, и **глобальные сети** (WAN, Wide Area Network), которые могут охватывать целый город, страну или даже весь мир. Наиболее важной разновидностью локальной сети является сеть Ethernet, поэтому мы рассмотрим ее в качестве примера локальной сети. Глобальные сети мы рассмотрим на примере сети Интернет, хотя технически Интернет представляет собой не единую сеть, а федерацию тысяч отдельных сетей. Однако в нашем случае можно рассматривать ее как единую глобальную сеть.

Ethernet

Классическая сеть Ethernet, описанная в стандарте IEEE Standard 802.3, состоит из коаксиального кабеля, к которому присоединены несколько компьютеров. Кабель называется **Ethernet** в честь *светоносного эфира*, по которому, как когда-то полагали, распространяются электромагнитные волны. (Когда английский физик девятнадцатого столетия Джеймс Кларк Максвелл обнаружил, что электромагнитное излучение может быть описано волновым уравнением, ученые предположили, что пространство должно быть заполнено неким эфирным носителем, колебаниями которого являются электромагнитные волны. Только после знаменитого эксперимента Михельсона и Морли в 1887 году, которым не удалось обнаружить эфир, физики поняли, что радиация может распространяться в вакууме.)

В первой версии сети Ethernet компьютер присоединялся к кабелю при помощи ответвителя, называемого **«зубом вампира»**, который буквально протыкал кабель иглой до середины. Схема сети Ethernet показана на рис. 8.28, *а*. Этим зубом было трудно попасть в жилу, поэтому ранее использовались настоящие соединители. Тем не менее электрически все компьютеры были соединены так, как если бы кабели их сетевых интерфейсных карт были спаяны вместе.

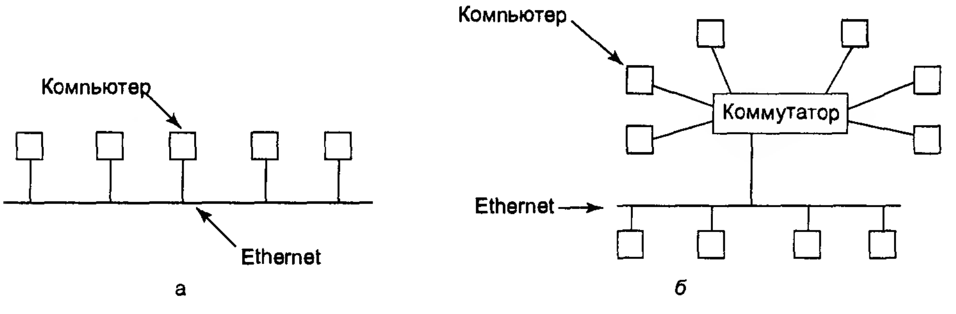


Рис. 8.28. Классическая сеть Ethernet (а); коммутируемая сеть Ethernet (*б*)

Чтобы послать пакет по сети Ethernet, компьютер сначала прослушивает кабель, определяя, не передает ли по нему какой-либо другой компьютер. Если кабель свободен, он начинает передавать пакет, состоящий из короткого заголовка, за которым

следует от 0 до 1500 байт полезной нагрузки. Если кабель занят, компьютер просто ждет, пока не завершится текущая передача, после чего начинает передавать сам.

Если два компьютера начинают передачу одновременно, происходит конфликт, который обнаруживается обоими компьютерами. При этом оба компьютера прекращают передачу и ожидают в течение случайного интервала времени от 0 до T мкс, после чего начинают все сначала. Если опять происходит столкновение пакетов, компьютеры ждут в течение случайного интервала времени от 0 до $2T$ мкс и т. д. При каждом следующем столкновении максимальный интервал ожидания удваивается, в результате чего вероятность следующего столкновения снижается. Такой алгоритм называется **двоичным экспоненциальным откатом**. Мы уже встречались с ним, когда рассматривали способы снижения накладных расходов при опросе мьютексов.

Длина кабеля и число компьютеров в сети Ethernet ограничены. Чтобы выйти за допустимые стандартом пределы, можно объединить несколько сетей Ethernet при помощи устройств, называемых **мостами**. Мост позволяет трафику перетекать из одной сети в другую, обеспечивая общение компьютеров, находящихся в разных сетях.

Во избежание проблем со столкновениями пакетов в современных сетях Ethernet применяются коммутаторы (рис. 8.28, *б*). У каждого коммутатора есть несколько портов, к которым можно присоединить компьютер, сеть Ethernet или другой коммутатор. Если пакету удается избежать столкновений и добраться до коммутатора, он там буферируется и пересылается через порт в направлении машины-получателя. Если каждому компьютеру выделить собственный отдельный порт, можно устранить все столкновения пакетов. Платой за это будут более громоздкие и дорогие коммутаторы. Возможны компромиссные решения, при которых к одному порту подключаются несколько компьютеров.

Интернет

Сеть Интернет появилась как развитие экспериментальной сети с коммутацией пакетов ARPANET, финансируемой управлением перспективного планирования научно-исследовательских работ (ARPA, Advanced Research Projects Agency) при Министерстве обороны США. Ее жизнь началась в декабре 1969 года с трех компьютеров в Калифорнии и одного в штате Юта. Цель данного проекта заключалась в создании высоконадежной сети, способной обеспечивать передачу военной информации даже в случае поражения большого числа отдельных участков сети в результате прямого попадания в них ядерных зарядов. При этом данная сеть должна была автоматически перенаправлять трафик в обход поврежденных узлов.

В 70-е годы сеть ARPANET быстро росла и в конечном итоге соединила сотни компьютеров. Наконец, к ней были присоединены радиосеть, спутниковая сеть и тысячи сетей Ethernet, в результате чего была сформирована федерация сетей, известная сегодня как Интернет.

Интернет состоит из компьютеров двух типов: хостов и маршрутизаторов. **Хостами** являются персональные компьютеры, лэптопы, палмтопы, серверы, мэйнфреймы и другие компьютеры, являющиеся собственностью компаний и частных лиц, желающих подключиться к Интернету. **Маршрутизаторы** — это специализированные коммутирующие компьютеры, принимающие приходящие пакеты

с одной или нескольких линий и отправляющие их дальше по одной или нескольким выходным линиям. Маршрутизатор подобен коммутатору (см. рис. 8.28, *б*), но также и отличается от него. Мы не станем обсуждать здесь отличия маршрутизатора от коммутатора. Маршрутизаторы объединяются в большие сети, в которых каждый маршрутизатор связан проводами или оптоволоконными кабелями с другими маршрутизаторами и хостами. Большие национальные и глобальные сети маршрутизаторов управляются телефонными компаниями и поставщиками услуг Интернета.

На рис. 8.29 показана схема части Интернета. Наверху мы имеем магистрали, как правило, управляемые магистральным оператором. Магистраль состоит из некоторого количества соединенных высокоскоростными оптоволоконными кабелями маршрутизаторов, связанных также с магистралями, управляемыми другими (конкурирующими) телефонными компаниями. Хосты, как правило, не присоединяются напрямую к магистралям, если не считать необходимые для управления и тестирования магистрали машины, управляемые телефонными компаниями.

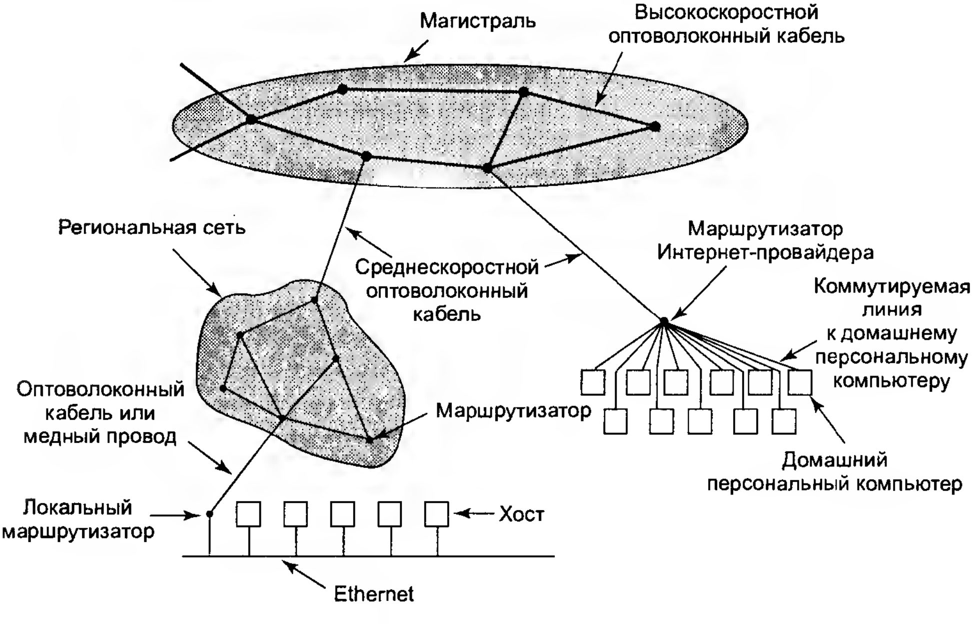


Рис. 8.29. Часть Интернета

К магистральным маршрутизаторам с помощью среднескоростных оптоволоконных кабелей присоединяются региональные сети и маршрутизаторы поставщиков услуг Интернета. К маршрутизаторам региональных сетей подключаются маршрутизаторы корпоративных локальных сетей Ethernet. Маршрутизаторы поставщиков услуг Интернета соединяются с модемными блоками, которыми пользуются клиенты Интернет-провайдеров. Таким образом, у каждого хоста в Интернете есть как минимум один путь, а часто много путей ко всем остальным хостам.

Весь трафик в Интернете посылается в виде пакетов. Каждый пакет содержит адрес пункта назначения, который используется при выборе маршрута. Когда пакет

попадает на маршрутизатор, маршрутизатор извлекает из него адрес получателя и ищет (часть) его в таблице, определяя, по какой выходной линии отправить этот пакет дальше следующему маршрутизатору. Эта процедура повторяется до тех пор, пока пакет не достигнет хоста-адресата. Таблицы маршрутизации в большой степени динамичны и постоянно обновляются по мере того, как маршрутизаторы и линии связи отключаются и снова включаются, а также при изменении состояния трафика.

Сетевые службы и протоколы

Все компьютерные сети предоставляют своим пользователям (хостам и процессам) определенные службы, реализуемые с помощью установленных правил, описывающих обмен сообщения. Ниже мы кратко рассмотрим эти темы.

Сетевые службы

Компьютерные сети предоставляют хостам и процессам, пользующимся ими, службы двух типов: ориентированные на соединение и без установления соединения. Модель **ориентированной на соединение службы** действует подобно телефонной системе. Чтобы поговорить с кем-нибудь, нужно поднять трубку, набрать номер, поговорить, после чего повесить трубку. Точно так же при пользовании службой на основе соединений сначала устанавливается соединение, используется, после чего служба разрывает связь. Существенным моментом подобного соединения является то, что оно действует подобно трубе: отправитель посылает объекты (биты) с одного конца, а получатель извлекает их на другом конце в том же самом порядке.

Службы без установления соединения, напротив, являются моделью почтовой системы. Каждое сообщение (письмо) содержит полный адрес назначения, и каждое сообщение пересылается по системе независимо от остальных. Обычно сообщение, отправленное первым, будет первым и получено. Однако в данной модели это не всегда так. Первое сообщение может по какой-либо причине задержаться в пути и прийти вторым. В модели служб на основе соединений подобное нарушение порядка невозможно.

Каждая служба характеризуется **качеством обслуживания**. Некоторые службы являются надежными в том смысле, что они никогда не теряют данные. Обычно надежная служба реализуется при помощи **подтверждений**, посылаемых получателем в ответ на каждое принятое сообщение, так что отправитель знает, дошло ли очередное сообщение или нет. Процесс пересылки подтверждений требует некоторых накладных расходов и снижает пропускную способность канала. Обычно подобные затраты не очень велики и окупаются, хотя иногда могут быть нежелательными.

Типичным примером необходимости надежной службы на основе соединений является пересылка файлов. Владелец файла хочет быть уверен, что все биты файла прибыли без искажений и в том же порядке, в котором были отправлены. Вряд ли кто отдаст предпочтение службе, которая случайным образом искажает информацию, даже если она значительно быстрее.

Надежные службы на основе соединений бывают двух типов: последовательности сообщений и байтовые потоки. В первом варианте сохраняются границы между сообщениями. Когда посылаются два сообщения размером по 1 Кбайт, то они прибывают в виде двух сообщений размером по 1 Кбайт и никогда как одно двухкилобайтное сообщение. При втором варианте связь представляет собой просто поток байтов, без разделения на отдельные сообщения. Когда 2 Кбайт прибывает к получателю, то нет способа определить, было ли это одно сообщение длиной 2 Кбайт, два сообщения длиной 1 Кбайт или же 2048 однобайтных сообщений. Если страницы книги посылаются по сети фотонаборной машине в виде отдельных сообщений, то, возможно, необходимо сохранить границы между сообщениями. С другой стороны, при регистрации с удаленного терминала в системе разделения времени вполне достаточно потока байтов с терминального компьютера.

Существуют системы, для которых задержки, связанные с пересылкой подтверждений, неприемлемы. В качестве примера такой системы можно назвать цифровую голосовую связь. В данном случае предпочтительнее допустить шумы на линии или искаженные слова, нежели большие паузы, вызванные отсылкой подтверждений и повторной передачей блоков данных.

Не все приложения требуют установки соединения. Например, при тестировании сети все, что требуется, — это способ переслать сообщение с высокой вероятностью его получения, но без гарантии. Ненадежная (то есть без подтверждений) служба без установления соединения часто называется **службой дейтаграмм** или дейтаграммной службой, по аналогии с телеграфной службой, также не предоставляющей подтверждений отправителю.

В других ситуациях бывает желательно отсутствие установления соединения для пересылки коротких сообщений, но надежность тем не менее существенна. Такая служба называется **службой дейтаграмм с подтверждениями**. Она подобна отправке заказного письма с подтверждением получения. Получив подтверждение, отправитель уверен, что письмо доставлено адресату, а не потеряно по дороге.

Кроме того, существует **служба запросов и ответов**, в которой отправитель посылает дейтаграммы, содержащие запросы, и получает ответы от получателя. Например, к данной категории можно отнести запрос к локальной библиотеке о том, где говорят по-уйгурски. Обычно модель запросов и ответов применяется для реализации общения в модели клиент-сервер: клиент посылает запрос, а сервер отвечает на него. Обсуждавшиеся выше типы служб сведены в таблицу на рис. 8.30.

	Служба	Пример
Ориентированный на соединение	Надежный поток сообщений	Последовательность страниц
	Надежный поток байт	Удаленная регистрация
	Ненадежное соединение	Цифровая голосовая связь
Без установления соединения	Ненадежная дейтаграмма	
	Дейтаграмма с подтверждением	Заказные письма
	Запрос-ответ	Запрос к базе данных

Рис. 8.30. Шесть типов служб

Сетевые протоколы

Во всех сетях есть узкоспециализированные правила, описывающие типы и форматы сообщений, которые могут посылаться в этих сетях, а также регламентирующие ответы на эти сообщения. Например, при некоторых обстоятельствах (скажем, перенос файла), когда сообщение посылается от источника адресату, адресат должен послать в ответ подтверждение правильного приема сообщения. В другой ситуации (например, при цифровой телефонии) подтверждения в ответ отправлять не требуется. Набор правил, с помощью которых общаются компьютеры, называется **протоколом**. Существует множество протоколов, включая протоколы для общения маршрутизатора с маршрутизатором, хоста с хостом и т. д. Детальный обзор компьютерных сетей и их протоколов см. в [323].

Все современные сети используют то, что называется **стеком протоколов**, то есть разные протоколы на разных уровнях. На разных уровнях протоколы занимаются различными вопросами. Так, на нижнем уровне протоколы описывают, как определить, где в битовом потоке начинается и заканчивается пакет данных. На более высоком уровне протоколы занимаются выбором маршрута пакета по сложным сетям от отправителя до получателя. А на еще более высоком уровне они гарантируют, что все пакеты сообщения прибыли правильно и в нужном порядке.

Так как большинство распределенных систем используют в качестве основы Интернет, ключевыми протоколами этих систем являются два главных протокола Интернета: IP и TCP. Протокол **IP** (Internet Protocol — Интернет-протокол) представляет собой дейтаграммный протокол, в котором отправитель вбрасывает в сеть дейтаграмму размером до 64 Кбайт и надеется, что она достигнет получателя. Никаких гарантий не предоставляется. Дейтаграмма может фрагментироваться по пути на пакеты меньшего размера. Эти пакеты перемещаются по Интернету независимо друг от друга, возможно, по разным маршрутам. Когда все пакеты достигают адресата, они собираются в нужном порядке и доставляются получателю.

В настоящий момент применяются две версии протокола IP: v 4 и v 6. На сегодня доминирует 4-я версия, поэтому мы опишем в данной книге ее, однако версия v6 получает все большую популярность. Каждый пакет протокола IP v 4 начинается с 40-байтового заголовка, содержащего, помимо других полей, 32-разрядный адрес отправителя и 32-разрядный адрес получателя. Эти адреса, называемые IP-адресами, образуют основу маршрутизации в Интернете. Как правило, они записываются четырьмя десятичными числами от 0 до 255, разделенными точками, например 192.31.231.65. Когда пакет прибывает на маршрутизатор, маршрутизатор извлекает из него адрес получателя и использует его для маршрутизации пакета.

Поскольку получение IP-дейтаграмм не подтверждается, для надежной связи в Интернете одного протокола IP недостаточно. Надежную связь обеспечивает другой протокол, обычно устанавливаемый поверх протокола IP. Это протокол **TCP** (Transmission Control Protocol — протокол управления передачей). Протокол TCP использует протокол IP для обеспечения ориентированных на соединение потоков. Чтобы воспользоваться протоколом TCP, процесс сначала устанавливает соединение с удаленным процессом. Требуемый процесс обозначается IP-адресом машины и номером порта на этой машине, который прослушивают процессы,

заинтересованные в получении входящего соединения. Когда соединение установлено, процесс просто передает по нему байты, которые гарантированно выходят с другого конца соединения неповрежденными и в правильном порядке. Реализация протокола TCP добивается этого при помощи порядковой нумерации, контрольных сумм и повторной передачи неверно полученных пакетов. Все это прозрачно для отправляющего и для получающего процессов. Они просто видят надежную межпроцессную связь, подобную каналу в системе UNIX.

Чтобы понять, как взаимодействуют данные протоколы, рассмотрим простейший случай пересылки очень маленького сообщения, для которого не требуется фрагментация ни на одном уровне. Хост находится в сети Ethernet, соединенной с Интернетом. Что происходит? Пользовательский процесс формирует сообщение и обращается к системному вызову, чтобы послать это сообщение по заранее установленному TCP-соединению. Стек протоколов ядра добавляет к началу сообщения TCP-заголовок, а затем IP-заголовок. Потом сообщение поступает к драйверу сети Ethernet, который добавляет свой заголовок, направляя, таким образом, пакет к маршрутизатору в сети Ethernet. Этот маршрутизатор посылает пакет по Интернету, как показано на рис. 8.31.

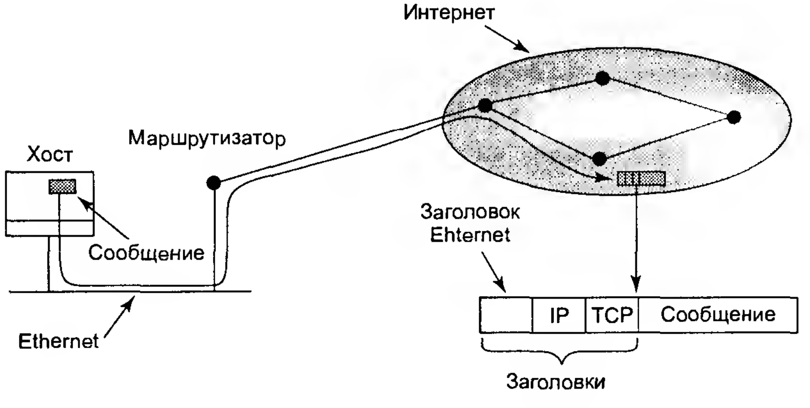


Рис. 8.31. Накапливание заголовков пакета

Для установки соединения с удаленным хостом (даже чтобы просто послать дейтаграмму) необходимо знать его IP-адрес. Так как пользоваться списками 32-разрядных IP-адресов неудобно для людей, была разработана схема, называемая **DNS** (Domain Name System — служба имен доменов). Она представляет собой базу данных, преобразующую имена хостов в формате ASCII в их IP-адреса. Данная система позволяет использовать DNS-имя *star.cs.vu.nl* вместо соответствующего ему IP-адреса 130.37.24.6. DNS-имена получили большую популярность благодаря электронной почте Интернета, в которой адреса имели форму user-name@DNS-host-name. При такой системе имен почтовая программа на передающем хосте ищет IP-адрес хоста-получателя в базе данных DNS, устанавливает TCP-соединение с почтовым демоном на этой машине и посылает сообщение в виде файла. Имя пользователя *user-name* отправляется вместе с письмом, чтобы указать, в какой почтовый ящик должно быть помещено письмо.

Промежуточное программное обеспечение, основанное на документе

Теперь, познакомившись с сетями и протоколами, мы можем начать поиск различных уровней промежуточного программного обеспечения, способных формировать непротиворечивую парадигму для приложений и пользователей. Начнем с простого и хорошо всем известного примера: Всемирной паутины (WWW, World Wide Web). Паутина была изобретена Тимом Бернерсом-Ли в 1989 году в Европейском центре ядерных исследований CERN (Conseil Europeen pour la Recherche Nucleaire) в Швейцарии. С тех пор это приложение распространилось по всему миру со скоростью лесного пожара.

Оригинальная парадигма приложения Всемирная паутина довольно проста: каждый компьютер может содержать один или несколько документов, называемых **web-страницами**. Каждая web-страница может содержать текст, изображения, звук, видео и т. д., а также **гиперссылки** (указатели) на другие web-страницы. Когда пользователь запрашивает web-страницу с помощью программы, называемой **web-браузером**, страница отображается на экране. Щелчок мышью на ссылке вызывает замещение текущей страницы на экране страницей, на которую указывает ссылка. Хотя в последнее время во Всемирную паутину было имплантировано множество всяческих погремушек, лежащая в основе парадигма остается неизменной: Паутина представляет собой большой направленный граф документов, которые могут содержать ссылки на другие документы (рис. 8.32).

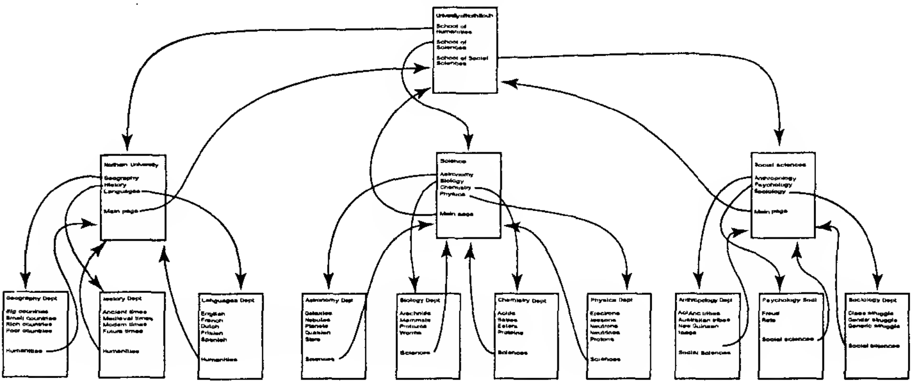


Рис. 8.32. Паутина представляет собой направленный граф документов

У каждой web-страницы есть уникальный адрес, называемый **URL** (Uniform Resource Locator — унифицированный указатель информационного ресурса), следующего вида: *протокол://DNS-имя/имя_файла*. Чаще всего применяется протокол *http* (HyperText Transfer Protocol — протокол передачи гипертекстовых файлов), но существуют также и другие протоколы, например, *ftp*. Следом за протоколом указывается DNS-имя хоста, содержащего файл. Наконец, указывается локальное имя файла, содержащего web-страницу.

В целом система работает следующим образом. Паутина представляет собой систему типа клиент-сервер, в которой пользователь является клиентом, а web-

сайт — сервером. Когда пользователь предоставляет браузеру URL, либо вводя его в окне, либо щелкая мышью на гиперссылке в текущей странице, браузер предпринимает определенные действия, чтобы достать запрашиваемую страницу. Например, пусть, запрашиваемый URL-указатель представляет собой http://www.acm.org/dl/faq.html. Чтобы получить эту страницу, браузер выполняет следующие действия:

1. Браузер запрашивает у DNS IP-адрес www.acm.org.
2. DNS отвечает: 199.222.69.151.
3. Браузер устанавливает TCP-соединение с портом 80 на хосте 199.222.69.151.
4. Затем браузер запрашивает файл dl/faq.html.
5. Сервер www.acm.org *посылает файл* dl/faq.html.
6. TCP-соединение разрывается.
7. Браузер отображает на экране текст dl/faq.html.
8. Браузер получает по сети и отображает на экране все изображения dl/faq.html.

В первом приближении такова основа Паутины. За десять с лишним лет ее существования были добавлены различные другие свойства, включая страницы стилей, динамические web-страницы, формируемые на лету, web-страницы, содержащие небольшие программы или сценарии, исполняющиеся на клиентской машине и т. д., но все это выходит за рамки данного обсуждения.

Промежуточное программное обеспечение, основанное на файловой системе

Основная идея Всемирной паутины заключается в том, что распределенная система должна выглядеть как гигантская коллекция документов, связанных гиперссылками. Второй подход состоит в том, чтобы придать распределенной системе вид огромной файловой системы. В данном разделе мы рассмотрим некоторые вопросы разработки всемирной файловой системы.

Использование модели файловой системы для распределенной системы означает, что имеется единая глобальная файловая система с пользователями по всему миру, способными читать и писать файлы, к которым у них есть доступ. Для связи процессов используется файловый обмен. Один процесс записывает данные в файл, а другой процесс считывает их оттуда. При таком подходе возникает множество стандартных вопросов, связанных с файловой системой, но также появляются и новые вопросы, относящиеся к распределению.

Модель переноса

Первый вопрос заключается в выборе между **моделью закачивания/скачивания** и **моделью удаленного доступа**. В первой модели, показанной на рис. 8.33, *а*, чтобы получить доступ к файлу, процесс сначала считывает его с удаленного сервера, на котором хранится этот файл. Если для файла разрешено только чтение, то файл читается локально для более высокой производительности. Если файл должен быть записан, он записывается также локально. Когда процесс заканчивает работу с файлом, обновленный файл отправляется обратно на сервер. В модели удален-

ного доступа файл остается на сервере, а клиент посылает серверу команды для выполнения работы на месте (рис. 8.33, *б*).

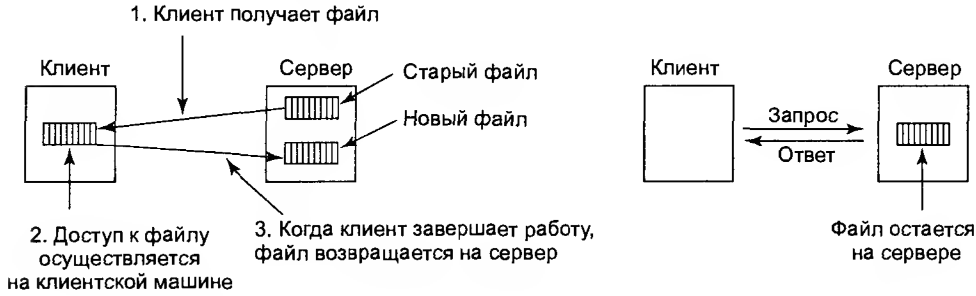


Рис. 8.33. Модель закачивания/скачивания (а); коммутируемая сеть Ethernet (*б*)

Преимущество модели закачивания/скачивания заключается в ее простоте и том факте, что перенос файла целиком эффективнее, чем перенос его по частям. К недостаткам данной модели относится необходимость наличия достаточно большого объема памяти для хранения файла целиком локально, к тому же перенос файла целиком, когда требуется только его часть, представляет собой излишние расходы. Наконец, при наличии нескольких конкурирующих пользователей возникает проблема непротиворечивости файлов.

Иерархия каталогов

Файлы представляют собой только часть этой картины. Другой ее частью является система каталогов. Все распределенные файловые системы поддерживают каталоги, содержащие многочисленные файлы. Следующий вопрос проектирования заключается в том, одинаково ли выглядит иерархия каталогов для всех клиентов. Поясним смысл этого вопроса с помощью рис. 8.34. Квадратами на рисунке изображены каталоги, а кружочками — файлы. На рис. 8.34, *а* показаны два файловых сервера, каждый из которых содержит три каталога и несколько файлов. На рис. 8.34, *б* мы видим систему, в которой для всех клиентов (и других машин) распределенная файловая система выглядит одинаково. Если путь */D/E/x* есть на одной машине, то он есть и на другой.

На рис. 8.34, *в*, напротив, разные машины видят файловую систему по-разному. Путь */D/E/x* может существовать на клиенте 1, но отсутствовать на клиенте 2. Для систем, работающих с удаленными серверами при помощи удаленного монтирования, ситуация на рис. 8.34, *в* является нормой. Такая схема обладает гибкостью и простотой реализации. Недостаток этого подхода заключается в том, что он не обеспечивает поведения всей системы как единой старомодной системы разделения времени. В системе разделения времени файловая система выглядит одинаково для всех процессов (рис. 8.34, *б*). Это облегчает программирование и работу в системе.

С этим вопросом близко связан такой вопрос, как наличие в системе глобального каталога, распознаваемого всеми машинами как корневого. Один из способов поддержки глобального корневого каталога состоит в том, чтобы создать корневой каталог, содержащий все серверы, и ничего, кроме серверов. При этом все

пути примут вид */сервер/путь*. У такого подхода имеются свои недостатки, но, по меньшей мере, пути будут одинаковыми для всей системы.

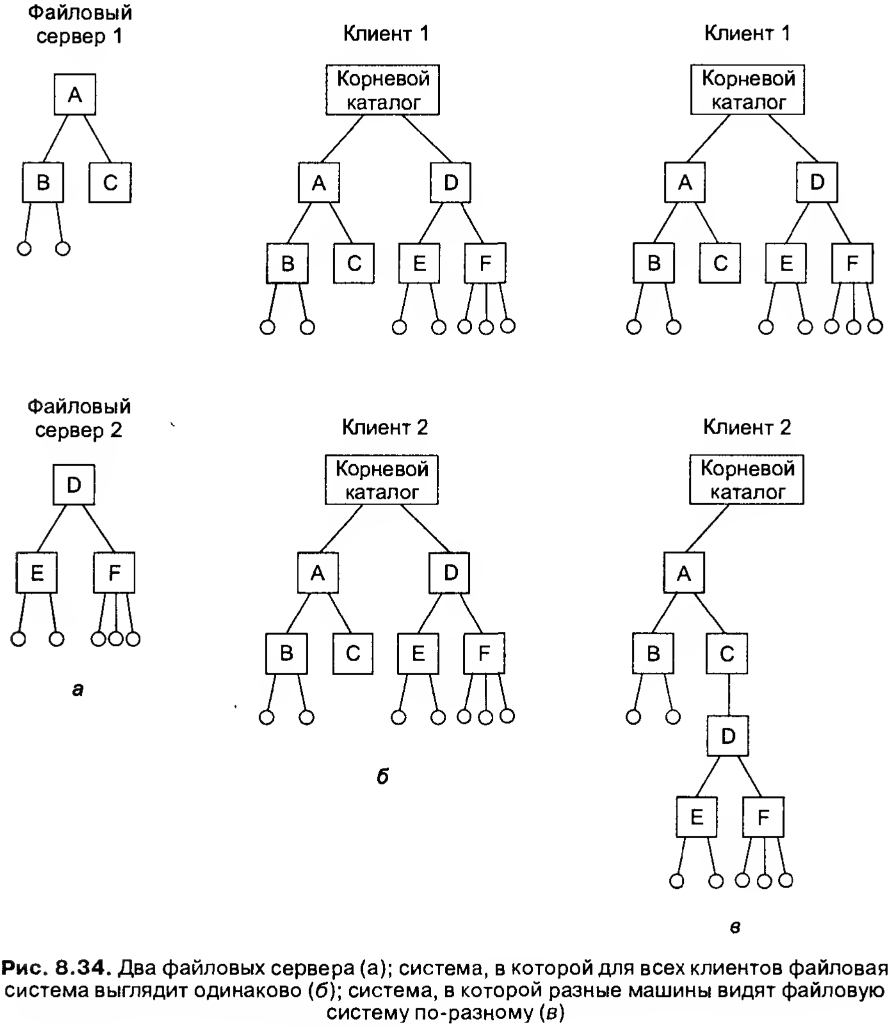


Рис. 8.34. Два файловых сервера (а); система, в которой для всех клиентов файловая система выглядит одинаково (*б*); система, в которой разные машины видят файловую систему по-разному (*в*)

Прозрачность именования

Главная проблема такого способа именования заключается в том, что она не полностью прозрачна. В данном контексте существует две формы прозрачности, которые следует различать. Первая форма, называемая **прозрачностью местоположения**, означает, что по имени пути невозможно определить расположение файла. По пути вида */server1/dir1/dir2/x* всем сразу становится ясно, что файл *x* находится на сервере *server1*, но при этом остается неизвестным, где расположен сам сервер. Сервер может перемещаться по сети без необходимости изменения имени пути. Это означает, что система обладает прозрачностью местоположения.

Однако предположим, что файл *x* крайне велик, а на сервере 1 (*server1*) мало дискового пространства. Более того, предположим, что на сервере 2 свободного пространства много. Было бы разумно предусмотреть в такой ситуации возможность автоматического перемещения файла *x* на сервер 2. К сожалению, когда имя сервера является первым компонентом всех путей, система не может переместить файл с одного сервера на другой, даже при наличии каталогов *dir1* и *dir2* на обоих серверах. Проблема заключается в том, что при перемещении файла с сервера 1 на сервер 2 путь файла также изменится с */server1/dir1/dir2/x* на */server2/dir1/dir2/x*. Программы, обращающиеся к этому файлу по пути, не смогут более находить этот файл. Про системы, в которых файлы могут перемещаться с одного сервера на другой без изменения пути файла, говорят, что они обладают **независимостью от местоположения**. Распределенные системы, указывающие имя машины или сервера в пути файла, определенно не обладают данным свойством. Системы, использующие монтирование устройств, также не обладают независимостью от местоположения, поскольку невозможно переместить файл из одной группы файлов (монтируемого модуля) в другую и продолжать при этом использовать старое имя пути. Независимость пути файла от его местоположения сложно реализовать, но это свойство представляет особую ценность для распределенных систем.

Подытоживая вышесказанное, в распределенных системах можно отметить три общих подхода к именованию файлов и каталогов:

1. Имя типа *машина + файл*, например, */машина/файл* или *машина:файл*.
2. Монтирование удаленной файловой системы в локальную файловую иерархическую структуру.
3. Единое пространство имен, одинаково выглядящее на всех машинах.

Первые два подхода легко реализуются, особенно как метод соединения существующих систем, при разработке которых не предполагалось их использование в качестве распределенных систем. Последний вариант сложен и требует детальной проработки, но значительно облегчает жизнь программистам и пользователям.

Семантика совместного использования файлов

Когда два или более пользователей вместе используют один и тот же файл, необходимо точно определить семантику чтения и записи файла, чтобы избежать возможных проблем. В однопроцессорных системах данные правила утверждают, что когда за системным вызовом `write` следует системный вызов `read`, то при чтении возвращается только что записанное значение (рис. 8.35, *а*). Аналогично, если системный вызов `read` следует за двумя последовательными системными вызовами `write`, считывается значение, записанное последним системным вызовом `write`. Таким образом, все системные вызовы упорядочиваются системой в единую последовательность, одинаковую для всех процессов. Про такую модель говорят, что она обладает **последовательной непротиворечивостью**.

В распределенной системе **последовательная непротиворечивость** может быть легко достигнута при условии, что имеется всего один файловый сервер, а клиенты не кэшируют файлы. Все системные вызовы `read` и `write` поступают напрямую на файловый сервер, который обрабатывает их в строгой последовательности.

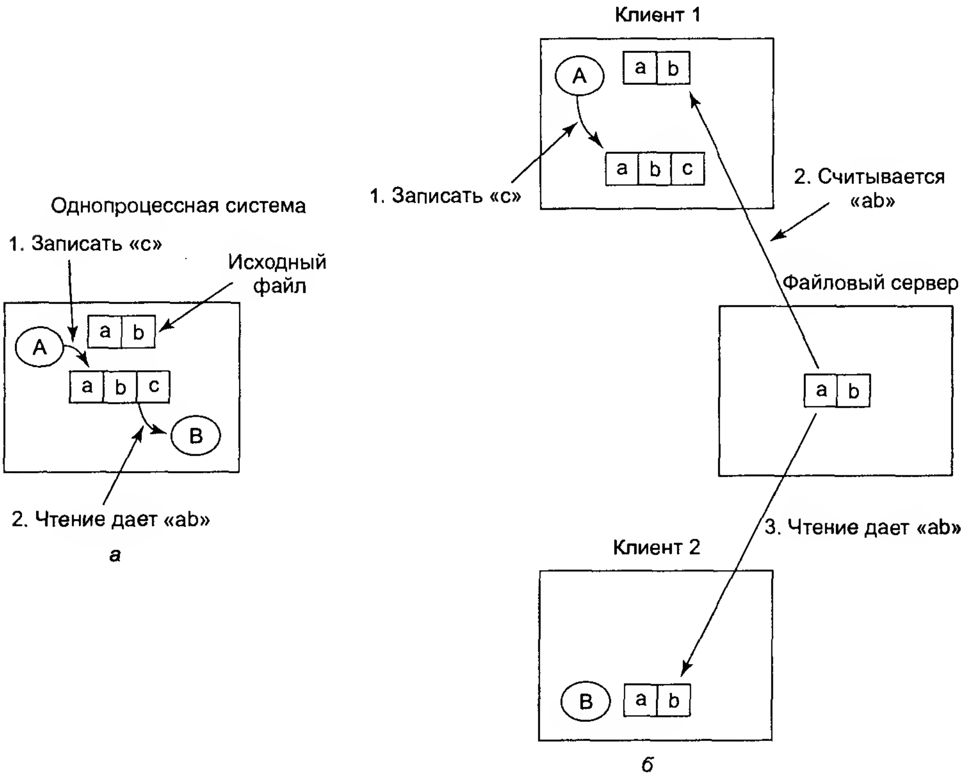


Рис. 8.35. Последовательная непротиворечивость (*а*); в распределенной системе с кэшированием при чтении файла можно получить его устаревшую копию (*б*)

Однако на практике производительность распределенной системы, в которой все файловые запросы должны выполняться на единственном сервере, как правило, невысока. Эта проблема часто решается за счет разрешения клиентам содержать собственные локальные копии активно используемых файлов в своих частных кэшах. Однако если клиент 1 модифицирует файл локально, а вскоре после этого клиент 2 прочитает этот файл с сервера, у второго клиента окажется устаревшая копия файла (рис. 8.35, *б*).

Один из способов разрешения данной проблемы заключается в том, чтобы немедленно распространять все изменения кэшируемых файлов обратно на серверы. Альтернативное решение представляет собой ослабление семантики совместного использования файлов. Вместо требования, чтобы при системном вызове read был виден эффект всех предшествовавших системных вызовов write, новое правило звучит так: «Изменения всех открытых файлов изначально видны только процессу, сделавшему эти изменения. Только когда файл закрывается, эти изменения становятся видны другим процессам». Принятие такого правила не изменит ситуацию на рис. 8.35, *б*, но благодаря ему такое поведение (процесс *B* получает оригинальную версию файла) будет теперь считаться правильным. Когда клиент 1 закрывает файл, он посылает на сервер измененную копию файла, поэтому после-

дующие системные вызовы read, как и требовалось, получат новую копию. Практически это все та же модель закачивания/скачивания, показанная на рис. 8.33. Данное семантическое правило широко применяется и называется **сеансовой семантикой**.

Использование сеансовой семантики поднимает вопрос о том, что произойдет, если два или более клиентов одновременно прочитают в кэш и модифицируют один и тот же файл. Одно решение заключается в том, что по мере того, как эти клиенты станут поочередно закрывать файл, его модифицированная копия каждый раз будет посылаться на сервер. «Победит» тот клиент, который закроет файл последним. В качестве варианта такой семантики можно принять правило, согласно которому окончательный вариант будет выбираться из нескольких кандидатов неким неопределенным способом.

Альтернативный подход к сеансовой семантике — использовать модель закачивания/скачивания, но автоматически блокировать скачанные файлы. Попытки всех остальных клиентов получить уже считанный кем-то файл будут приостанавливаться до того момента, пока первый клиент не вернет этот файл. Если данный файл пользуется повышенным спросом, сервер может посылать клиенту, удерживающему файл, сообщения с просьбой поторопиться, хотя клиент может и проигнорировать данные просьбы. Итак, составление корректной семантики использования общих файлов представляет собой непростое дело, в котором нет элегантных и эффективных решений.

Файловая система AFS

Некоторые системы промежуточного программного обеспечения, основанные на файловых системах, были построены и развернуты. Ниже мы кратко обсудим одну из таких систем, основанную на модели закачивания/скачивания (см. рис. 8.33, *а*). В главе 10 мы рассмотрим другую систему (NFS), основанную на модели удаленного доступа (см. рис. 8.33, *б*).

Файловая система **AFS** была разработана и реализована в университете Карнеги—Меллона [158, 239, 292]. Эта система была названа **Andrew File System** в честь двух первых спонсоров университета Эндрю Карнеги и Эндрю Меллона. Цель проекта, запущенного в начале 80-х годов, заключалась в обеспечении каждого студента и каждого сотрудника университета мощной рабочей станцией под управлением операционной системы UNIX, но с общей файловой системой. В данном случае файловая система использовалась в качестве системы промежуточного программного обеспечения, чтобы превратить множество рабочих станций в единую когерентную систему.

У каждого пользователя файловой системы AFS есть своя рабочая станция, на которой работает слегка модифицированная версия системы UNIX. Модификация заключается в добавлении к ядру программы, называемой **venus** (Венера), и запуске файлового сервера **vice** (вице-сервера) в пространстве пользователя. (Изначально программа venus также работала в пользовательском пространстве, но затем она была перемещена в ядро по соображениям производительности.) Расположение программ *venus* и *vice* показано на рис. 8.36, *а*. В административных целях рабочие станции пользователей группируются в ячейки. Ячейкой может быть локальная сеть, множество объединенных локальных сетей или даже целый факультет.

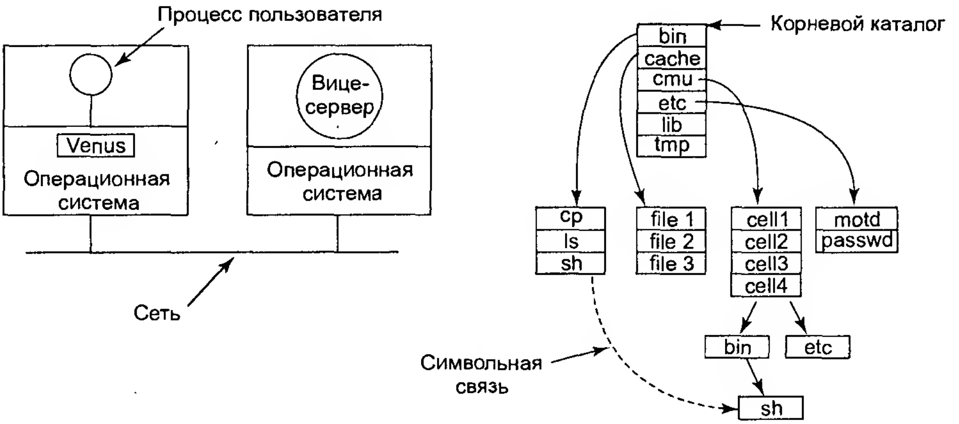


Рис. 8.36. Расположение программ venus и vice в файловой системе AFS (*а*); файловая система с точки зрения клиента (*б*)

Видимое программам пользователя пространство имен выглядит как традиционное дерево UNIX с добавлением каталогов */cmu* и */cache* (рис. 8.36, *б*). Каталог */cache* содержит имена кэшированных удаленных файлов. Каталог */cmu* содержит имена удаленных совместно используемых ячеек, под которыми располагаются их файловые системы. В результате удаленные файловые системы монтируются в каталоге */cmu*. Остальные каталоги и файлы являются локальными и не используются совместно. Разрешаются символьные ссылки от локальных файлов к совместно используемым файлам, как показано на примере файла *sh* на рис. 8.36, *б*.

Основная идея файловой системы AFS заключается в том, чтобы пользователь максимально применял потенциал локальной работы и, по возможности, минимально взаимодействовал с остальной частью системы. При открытии файла программа *venus* перехватывает системный вызов open и загружает файл целиком (или если файл очень велик, то большую его часть) на локальный диск в каталог */cache*. Дескриптор файла, возвращаемый системным вызовом open, ссылается на файл в каталоге */cache*, так что последующие системные вызовы read и write используют этот кэшированный файл.

Семантика, предлагаемая файловой системой AFS, близка к сеансовой семантике. Когда файл открывается, он получается с соответствующего сервера и помещается в каталог */cache* на локальном диске рабочей станции. Все операции чтения и записи выполняются с этой кэшированной копией файла. Когда файл закрывается, он закачивается обратно на сервер.

Однако для предотвращения возможности использования несвежих копий файла другими процессами программа *venus*, загружая файл в кэш, сообщает программе *vice*, следует ли отслеживать последовательные обращения других процессов к этому файлу. Если да, то вице-сервер записывает расположение кэшированного файла. Когда другой процесс системы открывает этот же файл, программа *vice* посылает программе *venus* соответствующее сообщение, после чего программа *venus* помечает элемент кэша как недействительный и, если копия файла была изменена, она копируется обратно на сервер.

Промежуточное программное обеспечение, основанное на совместно используемых объектах

Рассмотрим теперь третью парадигму. Вместо того чтобы считать все документами или файлами, будем называть все, что есть в системе, объектами. **Объект** — это набор переменных, объединенных вместе с набором процедур доступа к ним, называемых **методами**. Процессам не разрешается получать доступ к переменным напрямую. Вместо этого они должны вызывать методы.

CORBA

Некоторые языки программирования, такие как C++ и Java, являются объектно-ориентированными, но они имеют дело с объектами языкового уровня, а не с объектами времени исполнения. Одной из популярных систем, основанных на объектах времени исполнения, является технология **CORBA** (Common Object Request Broker Architecture — архитектура распределенных объектных приложений) [344]. Технология CORBA представляет собой систему типа клиент-сервер, в которой клиентский процесс может осуществлять операции с объектами, расположенными на серверах. Архитектура CORBA была разработана для неоднородной системы, состоящей из разнообразных аппаратных платформ и операционных систем, и программировалась на нескольких языках. Чтобы клиент на одной платформе мог вызвать сервер на другой платформе, между клиентом и сервером располагаются специальные программные посредники, называемые **ORB** (Object Request Broker — брокер объектных запросов). Посредники ORB играют в архитектуре CORBA важную роль. Благодаря им сама архитектура получила свое название.

Каждый объект системы CORBA описывается на специальном языке описания интерфейсов **IDL** (Interface Definition Language). В определении объекта указываются методы, экспортируемые объектом, и типы параметров для каждого метода. Спецификация IDL может быть откомпилирована в клиентскую процедуру-заглушку и храниться в библиотеке. Если клиентскому процессу заранее известно, что ему потребуется доступ к определенному объекту, этот объект компонуется вместе с объектным кодом клиентской заглушки. Спецификация IDL также может быть откомпилирована вместе со **скелетной** процедурой, используемой на серверной стороне. Если заранее нельзя сказать, какие объекты CORBA понадобятся процессу, возможно также применение динамического вызова, но принципы работы этого метода выходят за пределы данной книги.

При создании объекта CORBA также создается ссылка на этот объект, которая возвращается процессу, создавшему объект. В дальнейшем процесс обращается к методам созданного объекта по этой ссылке. Ссылка может быть передана другим процессам или сохранена в объектной библиотеке.

Чтобы вызвать метод объекта, клиентский процесс сначала должен получить ссылку на этот объект. Ссылку можно получить либо напрямую от создавшего объект процесса, либо при помощи поиска этого объекта или его метода по имени в специальном каталоге. Когда ссылка на объект получена, клиентский процесс упаковывает параметры вызываемого метода в стандартную структуру и связывается с клиентским ORB-посредником. Тот, в свою очередь, посылает сообщение серверному ORB-посреднику, который вызывает метод объекта. Весь механизм схож с вызовом удаленной процедуры RPC.

Цель ORB-посредников заключается в сокрытии всех низкоуровневых деталей связи и распределения от клиентской и серверной программ. В частности, ORB-посредники скрывают от клиента расположение сервера, аппаратное обеспечение сервера и операционную систему, работающую на сервере, информацию о том, является ли сервер двоичной программой или сценарием, является ли объект активным в настоящий момент, а также способ общения двух ORB-посредников (TCP/IP, RPC, общая память и т. д.).

В первой версии системы CORBA протокол между клиентским и серверным ORB-посредниками не был определен. В результате каждый разработчик ORB-посредников использовал свой отличный протокол, и различные реализации ORB не могли общаться друг с другом. Протокол был определен в версии 2.0. Для связи через Интернет используется протокол, называемый **IIOP** (Internet InterORB Protocol — протокол для связи между ORB-посредниками по Интернету).

Чтобы было можно использовать в системах CORBA объекты, не предназначенные специально для этого, каждый объект оснащается **адаптером объекта**. Это упаковщик, обрабатывающий такие рутинные операции, как формирование ссылок на объект и активация объекта в случае, если при вызове тот находится в неактивном состоянии. Организация всех этих частей системы CORBA показана на рис. 8.37. Детали системы CORBA показаны серым цветом.

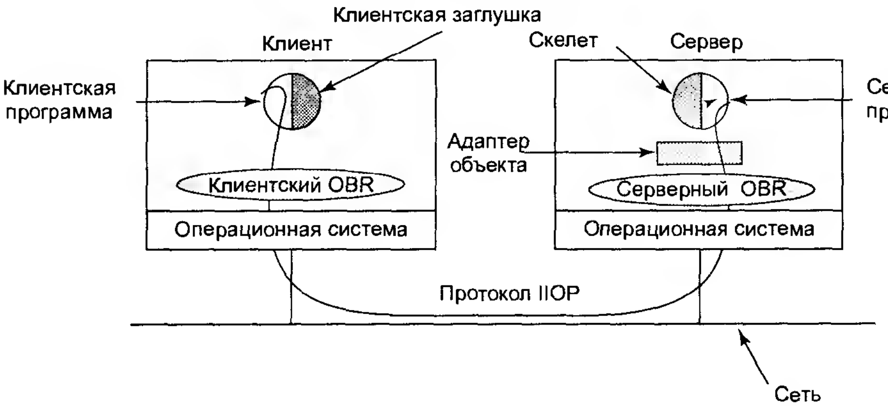


Рис. 8.37. Основные элементы распределенной системы, основанной на архитектуре CORBA

Серьезный недостаток системы CORBA заключается в том, что каждый объект расположен только на одном сервере, что означает крайне низкую производительность для объектов, одновременно используемых многими клиентами. На практике архитектура CORBA может приемлемо работать только в небольших системах, например, соединять процессы на одном компьютере, в пределах одной локальной сети или одной компании.

Система Globe

В качестве примера распределенной объектной системы, специально разработанной для поддержки миллиарда пользователей и триллиона объектов по всему миру, рассмотрим систему **Globe** [340, 341]. Для создания очень больших систем

используются две ключевые идеи. Первая из них — реплицируемые объекты. Если имеется единственная копия популярного объекта, доступ к которому хотят получить миллионы пользователей по всему миру, объект физически не сможет обработать все запросы. Подумайте об объекте, занимающемся биржевыми сводками или результатами спортивных соревнований. Репликация объекта позволяет распределить нагрузку по репликам.

Вторая ключевая идея заключается в гибкости. Во всемирной системе с миллиардом пользователей нет способа добиться всеобщего согласия по вопросу о выборе языка программирования, единой репликационной стратегии, общей модели безопасности или еще по какому-либо вопросу. Система должна позволять различное поведение различных пользователей и различных объектов и в то же самое время обеспечивать когерентную общую модель. Именно это и делает система Globe.

Необычность системы Globe заключается также в том, что, подобно технологии DSM, она основывается на модели распределенной совместно используемой памяти, но применительно ко всемирному контексту. В принципе использование нормальной технологии DSM, основанной на страницах, будет работать и в мировых масштабах, но вот производительность такой системы будет просто ужасной, поэтому в системе Globe применяется другой подход. Концептуально основная идея состоит в том, что мир полон объектов, каждый из которых содержит некое (скрытое) состояние и методы, обеспечивающие управляемый доступ к этому состоянию. Чтобы создать совместно используемую память в мировом масштабе, нужно запретить прямой доступ к внутреннему состоянию объекта при помощи команд `LOAD` и `STORE` и разрешить весь доступ только через методы. Так как объект системы Globe может одновременно активно использоваться многими процессами, он также называется **распределенным совместно используемым объектом**. Положение системы типа Globe показано на рис. 8.22, *в*.

Рассмотрим теперь реализацию масштабируемости и гибкости. У каждого объекта системы Globe есть объект класса, содержащий программы его методов. У каждого объекта также есть интерфейс или несколько интерфейсов, каждый из которых содержит пары (указатель на метод, указатель на состояние). Таким образом, имея интерфейс объекта, представляющий собой таблицу указателей, находящуюся в памяти во время исполнения, процесс может вызвать n-й метод, обратившись к процедуре, на которую указывает n-я пара в таблице интерфейса, и передав ей соответствующий указатель на состояние в качестве параметра. Указатель на состояние нужен на тот случай, если в памяти одновременно присутствуют, например, два объекта класса mailbox (почтовый ящик). У каждого объекта при этом есть собственный интерфейс и указатели состояния, но у обоих объектов общие указатели методов (рис. 8.38). В данном примере у процесса есть два открытых почтовых ящика, совместно использующих четыре метода, но каждый со своим собственным состоянием (сообщениями, сохраняемыми в экземпляре почтового ящика). Например, один почтовый ящик может быть для деловой переписки, а другой — для личной корреспонденции.

Хранение интерфейсов во время исполнения в таблицах в памяти означает, что объекты не ограничены использованием какого-либо одного языка. Такое реше-

ние было выбрано, так как во всемирной системе будет множество различных пользователей и программистов со своими языковыми предпочтениями. Методы объектов могут быть написаны на C, C++, Java или даже на ассемблере, если так пожелает владелец объекта. Цель интерфейсов состоит в том, чтобы скрывать от процессов то, что располагается за указателями методов. Такой комбинированный метод является более гибким, чем использование только одного языка программирования (например, только C++ или только Java), применяющийся в некоторых системах.

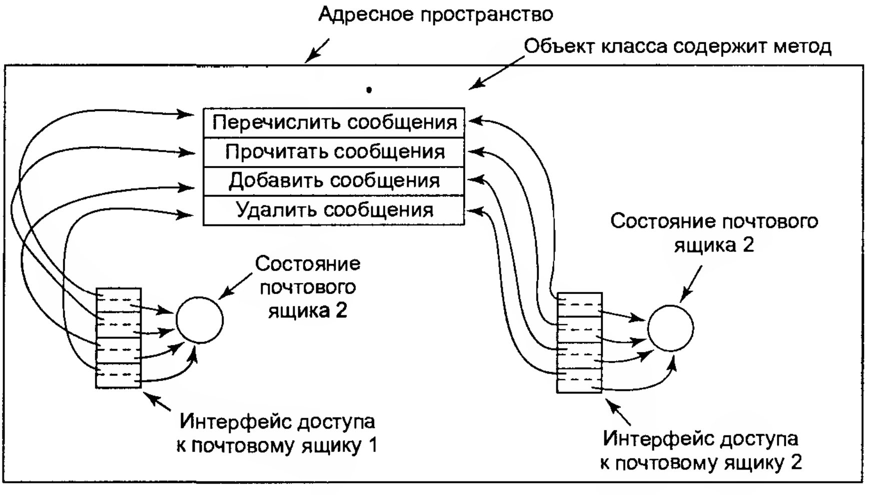


Рис. 8.38. Структура объекта Globe

Чтобы использовать объект Globe, процесс должен сначала связаться с ним, для чего этот объект нужно найти и определить хотя бы один адрес контакта (например, IP-адрес и порт). Проверка безопасности производится во время связывания, и если процессу разрешено связывать этот объект, объект класса (то есть программа) загружается в адресное пространство процесса, создается экземпляр класса и процессу возвращается указатель на его (стандартный) интерфейс. При помощи указателя на интерфейс процесс может вызывать методы, оперируя данным экземпляром объекта. В зависимости от объекта его состояние может быть состоянием по умолчанию или копией его текущего состояния, взятого у одной из уже существующих копий.

Представим себе простейший объект. У этого объекта есть целое число в качестве состояния и два метода: *read* (чтение) и *write* (запись), оперирующие с этим целым числом. Если несколько процессов в разных странах одновременно связываются с этим объектом, у всех них будет интерфейсная таблица, указывающая на объект класса, содержащий два метода (загружаемая во время связывания), как показано на рис. 8.39. У каждого процесса (потенциально) также есть копия целого числа, содержащего состояние объекта. Каждый метод *read* реализуется локально, но вызов метода *write* более сложен. Если объект хочет поддерживать последовательную непротиворечивость, для этого требуется специальный механизм.

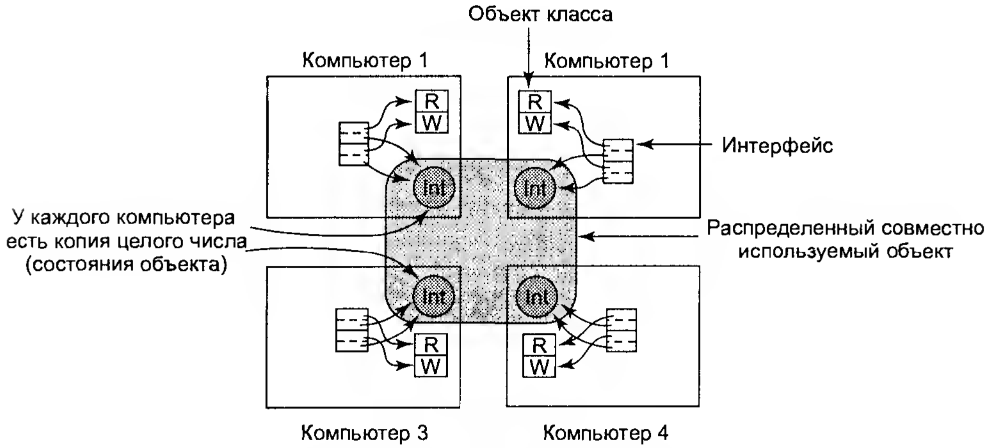


Рис. 8.39. Распределенный совместно используемый объект, состояние которого копируется на несколько компьютеров

Один из вариантов такого механизма заключается в наличии процесса, называемого **задатчиком последовательности**, чья работа состоит в выдаче последовательных номеров при обращении к нему. Чтобы выполнить операцию записи, метод *write* сначала получает последовательный номер, после чего при помощи многоадресной рассылки распространяет сообщение, содержащее этот порядковый номер, имя операции и параметр всем остальным процессам, связанным с этим объектом. Если два процесса одновременно вызовут метод *write*, они получат разные порядковые номера. Все процессы должны выполнять входящие методы в порядке номеров, а не в порядке их получения. Если процесс получит сообщение с порядковым номером 26, тогда как предыдущий порядковый номер был 24, он должен ждать сообщения с номером 25, прежде чем выполнить 26-е. Если 25-е сообщение не появится в течение определенного времени, процесс должен предпринять специальные действия по его обнаружению и получению. Такая схема гарантирует, что все операции записи выполнены в одинаковом порядке со всеми репликами объекта, обеспечивая последовательную непротиворечивость.

Технически схема вполне работоспособна, но не всем объектам нужна последовательная непротиворечивость. Рассмотрим, например, объект, содержащий биржевые цены. Если участник рынка ценных бумаг на бирже 1 издает команду обновления цен, а одновременно с ним участник рынка ценных бумаг на бирже 2 делает то же самое, то нет необходимости выполнять обновления всех копий этих объектов в том же порядке, так как они независимы. Вероятно, будет достаточно просто поместить все обновления в поток обновлений в том порядке, в котором они были посланы, но для этого не требуется единый задатчик последовательности. Достаточно использовать порядковые номера сообщений, проставляемые отправителями.

Приведенная выше схема репликации с равными копиями реплицированного объекта и разрешением обновления только после получения порядкового номера представляет собой лишь один из многих вариантов репликационных протоколов. В другом варианте хранится одна эталонная (главная) копия для каждого объекта

плюс несколько подчиненных копий. Все обновления посылаются эталону, который выполняет обновление и рассылает новое состояние всем подчиненным копиям.

Третья стратегия репликации объекта заключается в том, что состояние может быть только у одной копии объекта, тогда как все остальные копии являются заместителями объекта, не имеющими собственного состояния. Когда такой заместитель (например, клиентская машина) получает запрос метода *read* или *write*, он переадресовывает этот запрос копии, хранящей состояние, и запрос выполняется там.

Преимущество системы Globe состоит в том, что каждый объект может иметь свою стратегию репликации. Одни объекты могут использовать активную репликацию, тогда как другие объекты могут применять репликацию типа «главный-подчиненный» и т. д. Кроме того, у каждого объекта может быть своя стратегия обеспечения непротиворечивости, создания и удаления реплик, безопасности и т. п. Такая независимость достигается благодаря тому, что все стратегии реализуются внутри объекта. Пользователи объекта и администраторы системы даже не знают о стратегиях объекта. Этот подход радикально отличается от применяемого в системе CORBA, которая не скрывает стратегии, содержащиеся в объектах. В системе CORBA сложно создать 1000 объектов с 1000 различных стратегий.

Реализация объекта Globe показана на рис. 8.40. На рисунке изображены подобъекты, из которых состоит объект Globe. Управляющий объект принимает вызовы методов и использует другие подобъекты для их выполнения. Семантический подобъект выполняет работу, требуемую интерфейсом объекта. Программист, создающий объект, должен написать только часть программы объекта. Все остальное берется из стандартных библиотек, если только программист не хочет реализовать новую, еще не доступную стратегию. Репликационный подобъект занимается репликацией. Этот модуль может быть заменен, если вместо активной репликации желательно использовать репликацию типа «главный-подчиненный» или какую-либо другую репликационную стратегию, не затрагивая при этом остальной объект. Подобным же образом для реализации новой стратегии безопасности можно заменить подобъект безопасности, а для смены протокола (например, IP v4 на IP v6) можно заменить подобъект связи, и это не повлияет на остальные объекты.

Чтобы понять, как взаимодействуют данные подобъекты, рассмотрим, что получится, когда вызывается один из методов объекта. Программа, на которую указывает интерфейс, представляет собой управляющий подобъект, который просит подобъект репликации выполнить необходимые действия. Если объект реплицируется активно, сначала получается порядковый номер. Затем подобъект репликации велит всем репликам (включая собственную) выполнить работу, вызывая их семантические объекты. Если тип объекта «главный-подчиненный», а вызываемый метод находится на подчиненном объекте, то главному объекту посылается сообщение и т. д. В определенные моменты объект безопасности выполняет проверку (разрешен ли вызов метода, должны ли быть зашифрованы исходящие данные и т. д.).

Ключевым элементом системы Globe является **служба обнаружения**, позволяющая находить объект, где бы тот ни находился. Служба обнаружения строится в виде дерева, в узлах которого хранятся регистрационные данные объектов. Указатели на эти узлы передаются вверх по дереву, поэтому всегда можно найти сведения об объекте. Чтобы такая схема могла работать даже с мобильными объек-

тами, используются различные методы, включая группирование узлов дерева, кэширование и т. д. [20, 338, 339].

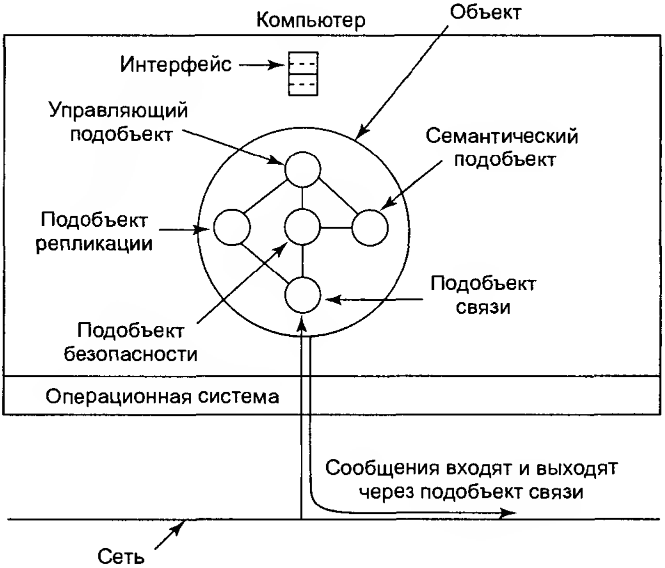


Рис. 8.40. Структура объекта Globe

Промежуточное программное обеспечение, основанное на координации

Последняя парадигма для распределенных систем называется **промежуточным программным обеспечением, основанным на координации**. Мы начнем изучение данной модели с системы Linda, научно-исследовательского проекта, давшего начало данному направлению, а затем рассмотрим два коммерческих примера, на которые система Linda оказала большое влияние: модель «публикация/подписка» и систему Jini.

Linda

Linda представляет собой новую систему связи и синхронизации, разработанную в Йельском университете Дэвидом Гелернтером и его студентом Ником Карьеро [55, 56, 125]. В системе Linda независимые процессы общаются через абстрактное **пространство кортежей**. Это пространство является глобальным по отношению ко всей системе, и процессы на любой машине могут вставлять кортежи в пространство кортежей или удалять их из него, независимо от того, как и где они хранятся. Для пользователя пространство кортежей выглядит как большая глобальная общая память, что мы уже видели ранее в различных формах (например, на рис. 8.22, *в*).

Кортеж представляет собой структуру языка С или запись языка Pascal. Он состоит из одного или нескольких полей, каждое из которых является значением

некоторого типа, поддерживаемого базовым языком. (Система Linda реализуется с помощью добавления библиотеки к стандартному языку, например к C.) В системе C-Linda типы полей включают целые числа, длинные целые числа, числа с плавающей точкой, а также составные типы, такие как массивы (включая строки) и структуры (но не другие кортежи). В отличие от объектов кортежи представляют собой чистые данные, они не содержат методов. В листинге 8.1 показаны три примера кортежей.

Листинг 8.1. Три примера кортежей системы Linda

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

С кортежами можно выполнять четыре операции. Первая из них, *out*, помещает кортеж в пространство кортежей. Например,

```
out("abc", 2, 5);
```

помещает кортеж ("abc", 2, 5) в пространство кортежей. Полями операции *out*, как правило, являются константы, переменные и выражения, как, например, команда

```
out("matrix-1", i, j, 3.14);
```

которая выводит в пространство кортежей кортеж ("matrix-1", i, j, 3.14), состоящий из четырех полей, второе и третье из которых определяются текущими значениями переменных *i* и *j*.

Кортежи можно получить из пространства кортежей при помощи примитива *in*. Они адресуются по содержимому, а не по имени или адресу. Поля примитива *in* могут быть выражениями или формальными параметрами. Рассмотрим, например, команду

```
in("abc", 2, ?i);
```

Эта операция «ищет» в пространстве кортежей кортеж, состоящий из строки "abc", целого числа 2 и третьего поля, содержащего любое целое число (предполагается, что *i* целое число). Если такой кортеж обнаруживается, он удаляется из пространства кортежей и переменной *i* присваивается значение третьего поля. Поиск соответствия и удаление из пространства кортежей представляют собой единую неделимую операцию, поэтому, если два процесса одновременно выполняют одинаковую операцию *in*, только для одного из них она завершается успешно (если только в пространстве кортежей не находится одновременно несколько кортежей, соответствующих искомому шаблону). В пространстве кортежей может содержаться даже несколько копий одного и того же кортежа.

Алгоритм проверки соответствия, используемый примитивом *in*, прост. Поля примитива, называемые **шаблоном**, сравниваются с соответствующими полями каждого кортежа в пространстве кортежей. Соответствие считается установленным при выполнении трех следующих условий:

1. У шаблона и кортежа одно и то же число полей.
2. Типы соответствующих полей совпадают
3. Каждая константа или переменная шаблона соответствует полю кортежа.

Формальные параметры, указанные знаком вопроса, за которым следует имя или тип переменной, не участвуют в сравнении (сравнивается только тип), но после успешного завершения операции поиска переменным присваивается соответствующее значение.

Если искомого кортежа найти не удается, запрашивающий процесс приостанавливается до тех пор, пока другой процесс не поместит в пространство кортежей нужный кортеж. Автоматическая блокировка и разблокирование процесса означает, что не имеет значения, которая операция будет выполнена раньше, операция ввода или вывода. Разница заключается только в наличии задержки, если ввод будет выполнен раньше вывода.

Блокирование процесса при отсутствии необходимого ему кортежа может использоваться по-разному. Например, с помощью этого свойства можно реализовать семафоры. Чтобы создать семафор или выполнить операцию up на семафоре *S*, процесс может выполнить команду

```
out("semaphore S");
```

А чтобы выполнить операцию down, процесс должен вызывать примитив

```
in("semaphore S");
```

Состояние семафора *S* определяется числом кортежей ("semaphore S") в пространстве кортежей. Если нет ни одного семафора, любая попытка получить его приведет к блокировке процесса, пока какой-либо другой процесс не создаст такой кортеж.

Помимо операций *out* и *in*, в системе Linda есть примитив *read*, который аналогичен примитиву *in*, но не удаляет кортеж из пространства кортежей. Также имеется примитив *eval*, оценивающий параметры и помещающий результат в виде кортежа в пространство кортежей. Этот механизм может использоваться для выполнения произвольных вычислений. С его помощью в системе Linda могут создаваться параллельные процессы.

Публикация/подписка

Наш следующий пример модели, основанной на координации, был создан под влиянием системы Linda и называется «**публикация/подписка**» [251]. Эта модель состоит из нескольких процессов, соединенных широковещательной сетью. Каждый процесс может производить информацию, потреблять информацию или и то и другое.

Когда у производителя информации есть новая информация (например, новые биржевые сводки), он рассылает ее всем по сети в виде кортежа. Это действие называется **публикацией**. Каждый кортеж содержит иерархически структурированную строку темы публикации с полями, разделенными точками. Процессы, которых интересуют определенные темы, могут подписаться на них. При этом можно использовать символ звездочки для обозначения всех разделов какой-либо темы. Чтобы подписаться на какую-либо тему, нужно сообщить ее специальному демону кортежей, работающему на той же машине, что и процесс.

Реализация модели «публикация/подписка» показана на рис. 8.41. Когда у процесса есть кортеж для публикации, он рассылает его с помощью широковещания

по локальной сети. Демон кортежей на каждой машине копирует все рассылаемые подобным образом кортежи в ОЗУ. Затем он просматривает строку темы сообщения, чтобы определить, которые процессы заинтересованы в получении этой информации, пересылая каждому такому процессу копию полученного сообщения. Кортежи также могут рассылаться по глобальным сетям или по Интернету. Для этого в каждой локальной сети одна машина должна исполнять роль информационного маршрутизатора, собирая все опубликованные кортежи и пересылая их другим локальным сетям, реализуя, таким образом, широковещание. При этом можно учитывать наличие в локальной сети-получателе хотя бы одного заинтересованного подписчика, дабы не занимать зря линии передачи. Для этого информационные маршрутизаторы должны обмениваться информацией о подписчиках.

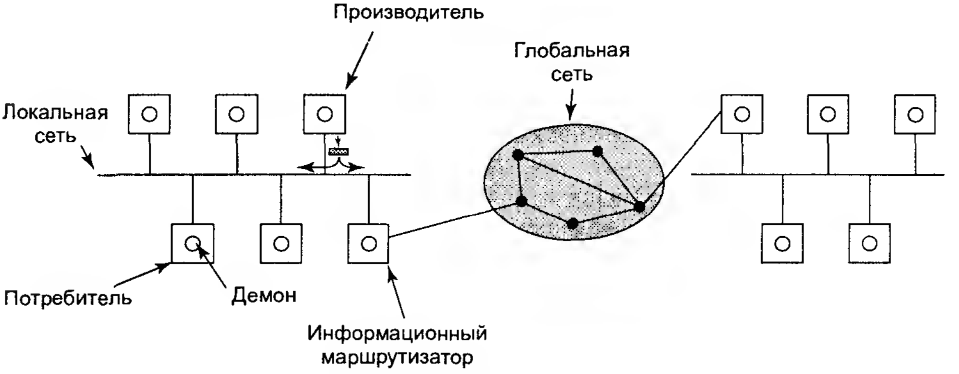


Рис. 8.41. Архитектура модели «публикация/подписка»

В данной модели могут быть реализованы различные типы семантики, включая надежную доставку и гарантированную доставку, даже в случае сбоев компьютеров. В последнем случае необходимо хранить старые кортежи на случай, если они понадобятся впоследствии. Они могут храниться в базе данных, которая подписывается на все темы кортежей. Это может быть выполнено при помощи упаковывания базы данных в адаптер, чтобы имеющаяся база данных могла работать в модели «публикация/подписка». По мере поступления кортежи перехватываются адаптером и помещаются в базу данных.

Модель «публикация/подписка» полностью отделяет производителей от потребителей, как и система Linda. Однако иногда бывает полезно наличие обратной связи. Для этого можно опубликовать кортеж с вопросом: «Кого интересует тема x?» Ответы поступают обратно в виде кортежей вида «Меня интересует тема x».

Jini

В течение 50 лет архитектура компьютеров была процессоро-центричной, то есть компьютер представлял собой автономное устройство, состоящее из центрального процессора, оперативной памяти и почти всегда устройства хранения информации большой емкости, то есть диска. Система Jini (произносится «джини», что звучит так же, как и genie, то есть джин, гений), созданная корпорацией Sun Microsystems,

явилась попыткой заменить сложившуюся модель моделью, в центре которой находится сеть.

Система Jini состоит из большого количества самодостаточных Jini-устройств, каждое из которых предлагает другим устройствам одну услугу или несколько видов услуг. Jini-устройство может быть установлено в сеть и мгновенно начать предоставление услуг без сложных процедур установки. Обратите внимание, что устройства устанавливаются не в *компьютер*, как в традиционном случае, а в *сеть*. Jini-устройство может быть традиционным компьютером, но также принтером, палмтопом, сотовым телефоном, телевизором, стереокомбайном или другим устройством с центральным процессором, оперативной памятью и (возможно, беспроводным) соединением с сетью. Система Jini представляет собой свободную федерацию Jini-устройств, которые могут входить в систему и выходить из системы по своему желанию, без централизованного управления.

Когда Jini-устройство хочет присоединиться к федерации, оно передает по локальной сети с помощью широковещания пакет с вопросом о наличии в данном районе **службы поиска**. Для нахождения службы поиска используется специальный **протокол обнаружения**, один из нескольких «зашитых» протоколов системы Jini. (В качестве альтернативы новое Jini-устройство может ждать, пока не придет одно из периодически рассылаемых объявлений службы поиска, но этот механизм не будет рассматриваться здесь.)

Когда служба поиска видит, что новое устройство хочет зарегистрироваться, она посылает в ответ программу, выполняющую регистрацию. Поскольку Jini представляет собой систему, целиком реализованную на языке Java, посылаемая программа также написана на языке Java, интерпретируемом виртуальной машиной Java (JVM, Java Virtual Machine). Каждое Jini-устройство должно уметь исполнять программы, написанные на языке Java, как правило, в режиме интерпретации. Затем новое устройство исполняет полученную программу, связывающуюся со службой поиска и регистрирующуюся в ней на некий установленный интервал времени. Пока не истек данный интервал времени, устройство может перерегистрироваться, если оно того пожелает. Такая схема означает, что Jini-устройство может покинуть систему, просто выключившись, и о существовании этого устройства система вскоре забудет. Таким образом, не требуется ни специальной процедуры выхода из системы, ни централизованного управления. Концепция регистрации на определенный срок называется получением **аренды**.

Обратите внимание, что поскольку программа регистрации устройства загружается в устройство по сети, эта программа может изменяться по мере развития системы, для чего не потребуется изменений аппаратного и программного обеспечения в самом устройстве. В самом деле, устройству даже не известен протокол регистрации. Оно знает только о части этого протокола, касающейся предоставления некоторых атрибутов и прокси-кода, которым в дальнейшем будут пользоваться другие устройства для доступа к устройству.

Устройство или пользователь, ищущие определенную службу, могут запросить о ней поисковую службу. При успешном запросе прокси, предоставленный устройством во время регистрации, отсылается запрашивающему устройству и запускается на нем для установления контакта. Таким образом, устройство или пользователь могут общаться с другим устройством, даже не зная его адреса и протокола.

Клиенты и службы (аппаратные или программные устройства) общаются и синхронизируются при помощи пространств **JavaSpaces**, спроектированных по образу пространства кортежей системы Linda, но обладающими существенными отличиями от него. Каждое пространство JavaSpace состоит из некоторого количества элементов строго определенных типов. Эти элементы напоминают кортежи системы Linda с тем отличием, что их тип жестко задан, тогда как кортежи системы Linda не имеют типов. Каждый элемент состоит из нескольких полей, каждое из которых имеет определенный тип языка Java. Например, элемент типа служащий может состоять из строки (для имени), целого числа (номер отдела), второго целого числа (телефон) и логической переменной, означающей, работает ли данный служащий полный рабочий день.

В пространстве JavaSpace определены всего четыре метода (хотя у двух из них есть подвиды):

1. **Write** (записать): поместить новый элемент в пространство JavaSpace.
2. **Read** (прочитать): копировать элемент, соответствующий шаблону из пространства JavaSpace.
3. **Take** (взять): копировать и удалить элемент, соответствующий шаблону из пространства JavaSpace.
4. **Notify** (уведомить): уведомить вызывающее устройство, когда в пространство JavaSpace записывается элемент, соответствующий шаблону.

Метод *write* принимает в качестве входных аргументов элемент и его срок аренды, то есть время, когда этот элемент должен быть уничтожен. В системе Linda, напротив, кортежи остаются в пространстве кортежей, пока их не удалят. В пространстве JavaSpace один и тот же элемент может содержаться несколько раз, так что пространство JavaSpace не является математическим множеством (как и пространство кортежей системы Linda).

Методам *read* и *take* необходимо задать в качестве параметра шаблон для поиска элемента. Каждое поле шаблона может содержать определенное указанное значение, которому должно соответствовать поле элемента или знак звездочки, означающий, что данное значение при поиске безразлично, а сравниваются только типы переменных. Если подходящий под шаблон элемент найден, он возвращается вызывающему устройству, а в случае метода *take* еще и удаляется из пространства JavaSpace. У каждого из этих методов JavaSpace есть два варианта, отличающихся поведением метода в случае отсутствия элемента, соответствующего шаблону. Один вариант метода немедленно возвращает сообщение об ошибке поиска, а другой ждет в течение определенного интервала времени (задаваемого в качестве параметра).

Метод *notify* регистрирует интерес в определенном шаблоне. Если когда-либо позднее появляется элемент, соответствующий шаблону, запускается метод *notify* вызывавшего устройства.

В отличие от пространства кортежей Linda, пространство JavaSpace поддерживает атомарные транзакции. С их помощью можно группировать несколько методов. Они либо все выполняются, либо не выполняется ни один из них. Во время транзакции изменения, производимые в пространстве JavaSpace, не видны вне транзакции. Только когда транзакция завершается, эти изменения становятся видны остальным пользователям.

Пространство JavaSpace может использоваться для синхронизации между общающимися процессами. Например, в ситуации «производитель-потребитель» производитель помещает элементы в пространство JavaSpace по мере их производства. Потребитель забирает их методом *take*, блокируясь, если нужного ему элемента нет. Пространство JavaSpace гарантирует атомарность выполнения каждого метода, поэтому невозможна ситуация, при которой процесс пытается прочитать элемент, созданный всего лишь наполовину.

Исследования в области многопроцессорных систем

В данной главе мы рассмотрели три типа многопроцессорных систем: мультипроцессоры, многомашинные системы и распределенные системы. Познакомимся теперь с исследованиями в данных трех областях. Большая часть исследований в области мультипроцессоров относится к аппаратному обеспечению, в частности к способам построения совместно используемой памяти и методам поддержания ее в согласованном состоянии. Тем не менее существуют и исследования в этой области, посвященные использованию мониторов виртуальной машины на мультипроцессорах [48] и управлению ресурсами мультипроцессоров [133]. Планирование потоков также составляет предмет исследований, как в плане алгоритмов планирования [16, 267], так и в плане обработки очереди запросов на запуск [84].

Создание многомашинных систем существенно проще построения мультипроцессоров. Все, что требуется, — это несколько персональных компьютеров или рабочих станций и высокоскоростная сеть. По этой причине многомашинные системы представляют собой популярную тему исследований в университетах. Большая часть работ посвящена распределенной памяти общего доступа в той или иной форме, как основанной на страничной организации, текущем каталоге и реализуемой чисто программно [57, 114, 163, 165, 294, 315]. Оптимизация связи на уровне пользователя [347] и балансировка нагрузки [144] также являются темами исследований.

Множество статей посвящено распределенным системам, например кэшированию web-страниц [360], распределенным файловым системам [8, 148, 329] и мобильным файловым системам [298].

Резюме

Производительность и надежность компьютерных систем могут быть существенно увеличены при использовании многопроцессорных систем. Тремя типами организации многопроцессорных систем являются мультипроцессоры, многомашинные системы и распределенные системы.

Мультипроцессор состоит из двух или более центральных процессоров, совместно пользующихся общей оперативной памятью. Эти центральные процессоры могут быть соединены шиной, координатным коммутатором или многоступенчатой коммутаторной сетью. Возможно применение различных конфигураций опе-

рационных систем, включая наличие собственной операционной системы у каждого центрального процессора, наличие одной главной операционной системы и нескольких подчиненных операционных систем или симметричного мультипроцессора с одной копией операционной системы, способной выполняться на любом центральном процессоре. В последнем случае для обеспечения синхронизации требуется использование блокировок. Когда блокировка недоступна, центральный процессор может ждать ее освобождения в цикле опроса или выполнить переключение контекста. Возможно применение различных алгоритмов планирования, включая разделение времени, разделение памяти и бригадное планирование.

Многомашинные системы также состоят из двух и более центральных процессоров, но у каждого из этих центральных процессоров есть своя собственная память. У них нет общей оперативной памяти, поэтому весь обмен информацией осуществляется при помощи передачи сообщений. В некоторых случаях на сетевой интерфейсной плате установлен свой процессор. В таких случаях необходимо тщательно организовать связь центрального процессора и процессора на плате во избежание конфликтов. Для связи на уровне пользователя на многомашинных системах часто применяется вызов удаленной процедуры, но распределенная память совместного доступа также может использоваться. Здесь важным вопросом является балансировка нагрузки процессов. Среди многочисленных применяемых для этого алгоритмов применяются такие, как алгоритмы, инициируемые отправителем, алгоритмы, инициируемые получателем, и алгоритмы торгов.

Распределенные системы представляют собой слабосвязанные системы, каждый узел которых является полноценным компьютером с полным набором периферийных устройств и собственной операционной системой. Часто такие системы распределены по большим территориям. Поверх операционной системы может устанавливаться промежуточное программное обеспечение, предоставляющее однородный уровень для взаимодействующих с ним приложений. Среди различных типов промежуточного программного обеспечения — документное, файловое, объектное и координационное. В качестве примера промежуточного программного обеспечения можно назвать такие системы, как Всемирная паутина, AFS, CORBA, Globe, Linda и Jini.

Вопросы

1. Можно ли систему сетевых новостей USENET или проект SETI@home считать распределенной системой? (Проект SETI@home использует несколько миллионов персональных компьютеров для анализа данных, получаемых с радиотелескопа с целью поиска внеземного разума.) Если да, к каким категориям, описанным на рис. 8.1, они относятся.
2. Что произойдет, если два центральных процессора на мультипроцессоре попытаются одновременно получить доступ к одному и тому же слову памяти?
3. Если центральный процессор при каждой команде совершает одно обращение к памяти, сколько понадобится центральных процессоров, работающих со скоростью 200 MIPS, чтобы насытить шину с частотой 400 МГц? Предположим, что для обращения к памяти требуется один цикл шины. Теперь

рассмотрите ту же задачу для системы, в которой используется кэширование, и частота попаданий кэша составляет 90 %. Наконец, какая потребуется частота попаданий кэша, чтобы той же шиной могли совместно пользоваться 32 центральных процессора, не перегружая шину?

4. Предположим, что порвется провод между коммутаторами 2A и 3B в сети омега (см. рис. 8.5). Кто от кого окажется отрезанным?
5. Как выполняется обработка сигнала в модели на рис. 8.7?
6. При обращении к системному вызову в модели на рис. 8.8 сразу после прерывания должна быть решена проблема, которой не было в модели на рис. 8.7. Какова природа этой модели и каковы способы ее решения?
7. Перепишите программу enter_region из листинга 2.2, используя чистое чтение, чтобы уменьшить пробуксовку системы, вызываемую применением команды TSL.
8. Действительно ли необходимы критические области в программных секциях в операционной системе SMP для предотвращения конфликтов или достаточно мьютексов в структурах данных?
9. При применении команды TSL для синхронизации мультипроцессора блок кэша, содержащий мьютекс, будет мотаться взад-вперед между центральным процессором, удерживающим блокировку, и центральным процессором, запрашивающим ее, если оба процессора изменяют содержимое блока. Для снижения шинного трафика запрашивающий центральный процессор выполняет команду TSL раз в 50 циклов шины, но центральный процессор, удерживающий блокировку, изменяет блок кэша между командами TSL. Если блок кэша состоит из 16 32-разрядных слов, для переноса каждого из которых требуется один цикл шины, а шина работает с частотой 400 МГц, какая часть пропускной способности шины съедается перемещением блока кэша взад-вперед?
10. В тексте предполагалось использование алгоритма двоичного экспоненциального отката между вызовами команды TSL для опроса блокировки. Также предлагалось использовать максимальное значение интервала ожидания между опросами. Будет ли алгоритм правильно работать без ограничения интервала ожидания?
11. Предположим, что у процессоров нет команды TSL для синхронизации мультипроцессора. Вместо нее предлагается использовать команду SWP, атомарно меняющую содержимое регистра и ячейки памяти. Может ли эта команда использоваться для обеспечения синхронизации мультипроцессора? Если да, то как? Если нет, то почему?
12. В данном задании вам предлагается сосчитать, какую нагрузку на шину оказывает спин-блокировка. Допустим, что выполнение каждой команды процессора занимает 5 нс. Когда команда выполнена, выполняются все необходимые циклы шины. Каждый цикл шины занимает дополнительно 10 нс. Какую часть пропускной способности шины потребляет процесс, выполняющий в цикле команду TSL, чтобы войти в критическую область? Предположим, что работает нормальное кэширование, поэтому сама выборка команды TSL не потребляет циклов шины.

13. Как было сказано, рис. 8.12 иллюстрирует среду разделения времени. Почему только один процесс (*A*) показан на рис. 8.12, *б*?
14. Родственное планирование снижает частоту промахов кэша. Снижается ли при этом также частота промахов TLB? А как насчет страничных прерываний?
15. Чему равен диаметр соединительной сети для каждой из топологий, изображенных на рис. 8.16? Учитывайте одинаково все транзитные участки, и между хостом и маршрутизатором, и между двумя маршрутизаторами.
16. Рассмотрите топологию двойного тора (см. рис. 8.16, *г*), расширенного до размера $k \times k$. Чему равен диаметр сети? *Подсказка*: рассматривайте четные и нечетные значения k отдельно.
17. Для измерения пропускной способности соединительной сети часто применяется метод бисекции. Для этого из сети удаляется минимальное количество связей, так чтобы сеть распалась на две части. Затем суммируется пропускная способность удаленных линий связи. Если способов разбиения сети несколько, выбирается тот, при котором эта сумма минимальна. Чему равна бисекционная пропускная способность соединительной сети, представляющей собой куб $8 \times 8 \times 8$, если пропускная способность каждой линии равна 1 Гбайт?
18. Представим себе многомашинную систему, в которой сетевой интерфейс работает в режиме пользователя, поэтому для перемещения данных из ОЗУ источника в ОЗУ приемника требуется всего три операции копирования. Предположим, что перемещение 32-разрядного слова через сетевой интерфейс занимает 20 нс, а сама сеть работает со скоростью 1 Гбит/с. Чему будет равна задержка пересылки 64-байтового пакета без учета времени копирования? Чему будет равна задержка с учетом времени копирования? Теперь рассмотрите случай, в котором требуются две дополнительные операции копирования: в ядро на передающей стороне и из ядра на принимающей стороне. Чему будет равна задержка в этом случае?
19. Повторите предыдущее задание для случая с тремя операциями копирования и с пятью операциями копирования, но на этот раз рассчитайте не время задержки, а пропускную способность.
20. Чем должна отличаться реализация примитивов `send` и `receive` для мультипроцессора с общей памятью и многомашинной системы и как это отразится на производительности?
21. При переносе данных из ОЗУ в сетевую интерфейсную плату может использоваться фиксация страницы. Предположим, что выполнение системных вызовов для фиксации и открепления страниц занимает 1 мкс. Копирование данных занимает 5 байт/нс с помощью DMA и 20 нс на байт при помощи программного ввода-вывода. Насколько большим должен быть пакет, чтобы использование фиксации страницы и использование DMA было оправданным?
22. Когда процедура вынимается из программы и переносится на другой компьютер, чтобы обращаться к ней в дальнейшем при помощи метода RPC (вызов удаленной процедуры), могут возникнуть некоторые проблемы. В тексте указывались четыре из них: указатели, массивы неизвестных размеров,

неизвестные типы параметров и глобальные переменные. Среди не обсуждавшихся вопросов — что произойдет, если удаленная процедура обратится к системному вызову. Какие проблемы может это вызвать и что можно сделать для их разрешения?

23. Когда в системе DSM происходит страничное прерывание, требуется найти нужную страницу. Назовите два возможных метода поиска страницы.
24. Рассмотрите примеры распределения процессоров на рис. 8.25. Предположим, что процесс *H* перемещен с узла 2 в узел 3. Чему теперь равен суммарный внешний трафик?
25. Некоторые многомашинные системы позволяют работающим процессам мигрировать с одного узла на другой. Достаточно ли для этого просто остановить процесс, сохранить его образ в памяти и переслать этот образ на другой узел? Назовите две нетривиальные проблемы, которые следует решить, чтобы выполнить эту работу.
26. Почему существует ограничение на длину кабеля в сети Ethernet?
27. На рис. 8.27 третий и четвертый уровни помечены как промежуточное программное обеспечение и приложение на всех четырех машинах. В чем их сходство и различие для всех четырех платформ?
28. В таблице на рис. 8.30 перечислены шесть типов служб. Какая служба наиболее всего соответствует каждому из следующих приложений?
    1. Интерактивное видео по Интернету.
    2. Загрузка web-страницы.
29. DNS-имена имеют иерархическую структуру, например *cs.uni.edu* или *sales.general-widget.com*. Один из способов реализации DNS-имен может заключаться в поддержании централизованной базы данных, однако этот метод не применяется, так как такая база данных будет получать слишком большой поток запросов. Предложите свой метод реализации базы данных DNS.
30. При обсуждении метода обработки браузером URL-указателей утверждалось, что соединения устанавливаются через порт 80. Почему?
31. Могут ли URL-указатели, используемые в Паутине, обладать свойством прозрачности местоположения? Аргументируйте свой ответ.
32. Когда браузер получает web-страницу, он сначала устанавливает TCP-соединение, чтобы получить текст страницы (на языке HTML). Затем он разрывает соединение и изучает страницу. Если она содержит графические изображения, он снова создает отдельные TCP-соединения для каждого изображения. Предложите две альтернативные схемы для улучшения производительности.
33. При использовании сеансовой семантики изменения файлов немедленно становятся видимыми процессу, инициировавшему эти изменения, и никогда не видны процессам на других машинах. Однако вопрос о том, должны ли эти изменения становиться сразу же видимыми другим процессам на той же машине, остается открытым. Приведите аргументы в пользу обоих вариантов ответа.

34. В файловой системе AFS на клиентских машинах файлы кэшируются целиком. Предположим, что дисковое пространство, выделенное под кэшируемые файлы, переполнено. Что следует делать при запросе нового файла? Создайте соответствующий алгоритм.
35. В чем преимущества объектного доступа к данным перед совместно используемой памятью в ситуации, когда нескольким процессам требуется доступ к данным?
36. При выполнении операции *in* для обнаружения кортежа в системе Linda линейный поиск по всему пространству кортежей крайне неэффективен. Разработайте метод организации пространства кортежей, который ускорит выполнение операций *in*.
37. Для копирования буфера требуется время. Напишите программу на языке C, вычисляющую данное время для системы, к которой у вас есть доступ. Используйте функции *clock* или *times*, чтобы определить, сколько времени занимает копирование длинного массива. Поэкспериментируйте с массивами различного размера, чтобы отделить время копирования от накладных расходов.
38. Напишите на языке C функции, которые могли бы использоваться в качестве клиентского и серверного суррогатов для выполнения RPC-вызова стандартной функции *printf*, а также головной модуль для тестирования этих функции. Клиент и сервер должны общаться при помощи структуры данных, передаваемой по сети. Вы можете наложить разумные ограничения на длину строки формата и число, типы и размеры переменных, которые будет принимать ваш клиентский суррогат.
39. Напишите две программы для моделирования балансирования нагрузки в многомашинной системе. Первая программа должна назначать *m* процессов *n* машинам, в соответствии с файлом инициализации. Каждый процесс должен работать в течение случайного периода времени, величина которого распределена по Гауссу, причем среднее линейное и среднеквадратическое отклонение задаются в виде параметров модели. В конце работы каждый процесс создает несколько новых процессов, количество которых является случайной величиной, распределенной по Пуассону. После завершения работы каждого процесса центральный процессор должен решить, отдать свои процессы другому центральному процессору или попытаться самому найти новые процессы. Первая программа должна использовать алгоритм, инициируемый отправителем, чтобы избавиться от лишней работы, если у центрального процессора оказывается более *k* процессов. Вторая программа должна использовать алгоритм, инициируемый получателем, чтобы получать работу, когда это необходимо. Вы можете делать любые разумные допущения, но должны заявлять о них открыто и четко.

Глава 9
Безопасность

Многие компании обладают ценной информацией, которую они тщательно охраняют. Эта информация может быть технической (например, архитектура новой микросхемы или программного обеспечения), коммерческой (исследования конкурентоспособности или маркетинговые планы), финансовой (планы биржевых операций), юридической (документы о потенциальном слиянии или разделе фирм) и т. д. Часто эта информация защищается при помощи охранника в униформе, стоящего у входа в здание и проверяющего у всех входящих в здание наличие определенного значка. Кроме того, многие офисы и картотечные шкафы могут запираться на ключ, чтобы гарантировать доступ к информации только авторизованных сотрудников.

По мере того как возрастают объемы информации, хранящейся в компьютерных системах, необходимость в защите информации становится все важнее. Таким образом, защита информации от несанкционированного доступа является главной заботой всех операционных систем. К сожалению, достижение этой цели становится все более сложной задачей, так как стремление операционных систем к раздуванию все чаще воспринимается как нормальное и приемлемое явление. В следующих разделах мы рассмотрим различные вопросы, связанные с безопасностью и защитой; некоторые из них аналогичны вопросам защиты информации реального мира, хранящейся в виде обычных бумажных документов, а другие являются уникальными для компьютерных систем. В данной главе мы изучим вопрос компьютерной безопасности в применении к операционным системам.

Понятие безопасности

Термины «безопасность» и «защита» иногда смешиваются. Тем не менее часто бывает полезно провести границу между общими проблемами, связанными с гарантированием того, что файлы не читаются и не модифицируются неавторизованными лицами, с одной стороны, и специфическими механизмами операционной системы, используемыми для обеспечения безопасности, с другой стороны. Чтобы избежать путаницы, мы будем применять термин **безопасность** для обозначения общей проблемы и термин **механизмы защиты** при описании специфических механизмов операционной системы, используемых для обеспечения информационной безопасности в компьютерных системах. Однако граница между этими двумя терминами определена не четко. Сначала мы познакомимся с вопро-

сами безопасности, чтобы понять природу проблемы. Затем мы рассмотрим механизмы защиты и модели, способствующие обеспечению безопасности.

Проблема безопасности многогранна. Тремя ее наиболее важными аспектами являются природа угроз, природа злоумышленников и случайная потеря данных. Все эти вопросы будут поочередно рассмотрены в этой главе.

Угрозы

С позиций безопасности у компьютерной системы в соответствии с наличествующими угрозами есть три главные задачи, показанные в табл. 9.1. Первая задача, **конфиденциальность данных**, заключается в том, что секретные данные должны оставаться секретными. В частности, если владелец некоторых данных решил, что эти данные будут доступны только определенному кругу лиц, система должна гарантировать, что к этим данным не смогут получить доступ лица за пределами установленного круга. Как минимум владелец данных должен иметь возможность указать список пользователей, которым разрешено видеть ту или иную информацию, а система должна обеспечить исполнение этих требований.

Таблица 9.1. Задачи безопасности и угрозы безопасности

Задача	Угроза
Конфиденциальность данных	Демонстрация данных
Целостность данных	Порча или подделка данных
Доступность системы	Отказ обслуживания

Вторая задача, **целостность данных**, означает, что неавторизованные пользователи не должны иметь возможности модифицировать данные без разрешения владельца. Модификация данных в данном контексте означает не только изменение данных, но также их удаление или добавление фальшивых данных. Если система не может гарантировать, что хранящиеся в ней данные останутся неизменными до тех пор, пока владелец не решит их изменить, то такая система немногого стоит.

Третья задача, **доступность системы**, означает, что никто не может вывести систему из строя. Атаки типа **отказ в обслуживании** становятся все более распространенными. Например, если компьютер является сервером Интернета, он может быть затоплен мощным потоком запросов, при этом все его процессорное время уйдет на изучение входящих запросов. Так, если обработка запроса чтения web-страницы занимает 100 мкс, то любой пользователь, способный послать 10 000 запросов в секунду, может ликвидировать сервер. Существует множество моделей и технологий для противостояния атакам конфиденциальности и целостности данных. Отбивание атак типа «отказ обслуживания» является значительно более сложной задачей.

Еще один аспект проблемы безопасности касается права пользователя на конфиденциальность личной информации. Это довольно запутанная тема, связанная с различными юридическими и моральными вопросами. Должно ли и имеет ли право правительство собирать досье на всех и каждого, чтобы поймать лиц, уклоняющихся от налогов или получающих незаконные пособия? Должна ли полиция

иметь возможность слежки за всем и вся, пытаясь победить организованную преступность? Есть ли права доступа к частной информации у работодателей и страховых компаний? Что происходит, когда эти права вступают в конфликт с индивидуальными правами граждан? Все эти вопросы имеют чрезвычайно большое значение, но они находятся за пределами рассмотрения данной книги.

Злоумышленники

Большинство людей соблюдают закон, поэтому зачем беспокоиться о безопасности? Все дело в том, что, к сожалению, некоторые люди не столь добродетельны и желают доставить другим неприятности (например, с целью получения собственной коммерческой выгоды). В литературе по безопасности человека, сующего свой нос в чужие дела, называют **злоумышленником** и иногда **неприятелем**. Злоумышленники подразделяются на два вида. Пассивные злоумышленники просто пытаются прочитать файлы, которые им не разрешено читать. Активные злоумышленники пытаются незаконно изменить данные. При разработке защиты системы от злоумышленников важно представлять себе разновидности злоумышленников, от которых предстоит защищать систему. Наиболее распространенными категориями злоумышленников являются:

1. Случайные любопытные пользователи, не применяющие специальных технических средств. У многих людей есть компьютеры, соединенные с общим файловым сервером. И если не установить специальной защиты, благодаря естественному любопытству многие люди станут читать чужую электронную почту и другие файлы. Например, во многих системах UNIX новые, только что созданные файлы по умолчанию доступны для чтения всем желающим.
2. Члены организации, занимающиеся шпионажем. Студенты, системные программисты, операторы и другой технический персонал часто считает взлом системы безопасности локальной компьютерной системы личным вызовом. Как правило, они имеют высокую квалификацию и готовы посвящать достижению поставленной перед собой цели значительное количество времени.
3. Те, кто совершают решительные попытки личного обогащения. Некоторые программисты, работающие в банках, предпринимали попытки украсть деньги у банка, в котором они работали. Использовавшиеся схемы варьировались от изменения способов округления сумм в программах, для сбора, таким образом, с миру по нитке, до шантажа («Заплатите мне, или я уничтожу всю банковскую информацию»).
4. Занимающиеся коммерческим и военным шпионажем. Шпионаж представляет собой серьезную и хорошо финансированную попытку конкурента или другой страны украсть программы, коммерческие тайны, ценные идеи и технологии, схемы микросхем, бизнес-планы и т. д. Часто такие попытки включают подключение к линиям связи или установку антенн, направленных на компьютер для улавливания его электромагнитного излучения.

Очевидно, что попытка предотвратить кражу военных секретов враждебным иностранным государством отличается от противостояния попыткам студентов

установить забавные сообщения в систему. Необходимые для поддержания секретности и защиты усилия зависят от предполагаемого противника.

Еще одной разновидностью угрозы безопасности, появившейся в последние годы, является вирус. Вирусам будет уделено особое внимание в данной главе. В общем случае вирус представляет собой программу, реплицирующую саму себя и (как правило) причиняющую тот или иной ущерб. В определенном смысле автор вируса также является злоумышленником, часто обладающим высокой квалификацией. Основное различие между обычным злоумышленником и вирусом состоит в том, что первый из них представляет собой человека, лично пытающегося взломать систему с целью причинения ущерба, тогда как вирус является программой, написанной таким человеком и выпущенной в свет с надеждой на причинение ущерба. Злоумышленники пытаются взломать определенные системы (например, сеть какого-либо банка или Пентагона), чтобы украсть или уничтожить определенные данные, тогда как вирус обычно действует не столь направленно. Таким образом, злоумышленника можно уподобить наемному убийце, пытающемуся уничтожить конкретного человека, в то время как автор вируса больше напоминает террориста, пытающегося убить большое количество людей, а не кого-либо конкретно.

Случайная потеря данных

Помимо различных угроз со стороны злоумышленников, существует опасность потери данных в результате несчастного случая. К наиболее распространенным причинам случайной потери данных относятся:

1. Форс-мажор: пожары, наводнения, землетрясения, войны, восстания, крысы, изгрызшие ленты или гибкие диски.
2. Аппаратные и программные ошибки: сбои центрального процессора, нечитаемые диски или ленты, ошибки при передаче данных, ошибки в программах.
3. Человеческий фактор: неправильный ввод данных, неверные установленные диск или лента, запуск не той программы, потерянные диск или лента и т. д.

Большая часть этих проблем может быть разрешена при помощи своевременного создания соответствующих резервных копий, хранимых на всякий случай вдали от оригинальных данных. Хотя проблема защиты информации от случайных потерь кажется пустяковой по сравнению с задачей противостояния умным злоумышленникам, на практике больше ущерба наносят именно несчастные случаи.

Основы криптографии

Некоторые сведения о криптографии могут оказаться полезными для понимания разделов этой главы и некоторых следующих. Однако серьезное обсуждение криптографии выходит за рамки этой книги. Эта тема подробно рассматривается во многих прекрасных трудах по компьютерной безопасности. Читатели, интересующиеся данным вопросом, могут прочитать, например, [176, 266]. Ниже будет дан очень краткий обзор вопроса криптографии для читателей, совсем незнакомых с данной темой.

Задача криптографии заключается в том, чтобы взять сообщение или файл, называемый **открытым текстом**, и преобразовать его в **зашифрованный текст** таким образом, чтобы только посвященные могли преобразовать его обратно в открытый текст. Для всех остальных зашифрованный текст должен представлять собой просто непонятный набор битов. Это может показаться странным для новичков в этой области, но алгоритмы (функции) шифрации и дешифрации всегда должны быть открытыми (публичными). Попытки удерживать их в секрете никогда не увенчиваются успехом и всего лишь вводят в заблуждение людей, пользующихся данными алгоритмами, создавая ложную иллюзию защищенности данных. В промышленности такая тактика, называемая **секретностью благодаря неизвестиости**, применяется только новичками в области безопасности. Как ни странно, к этой категории относятся многие крупные многонациональные корпорации, которым следовало бы иметь лучшее представление о данном вопросе.

На самом деле секретность зависит от параметров алгоритмов, называемых **ключами**. Мы будем использовать формулу $C = E(P, K_E)$, обозначающую, что при зашифровке открытого текста P с помощью ключа K получается зашифрованный текст C.

Аналогично формула $P = D(C, K_D)$ означает расшифровку зашифрованного текста C для восстановления открытого текста. Схематично процессы шифрования и дешифровки показаны на рис. 9.1.

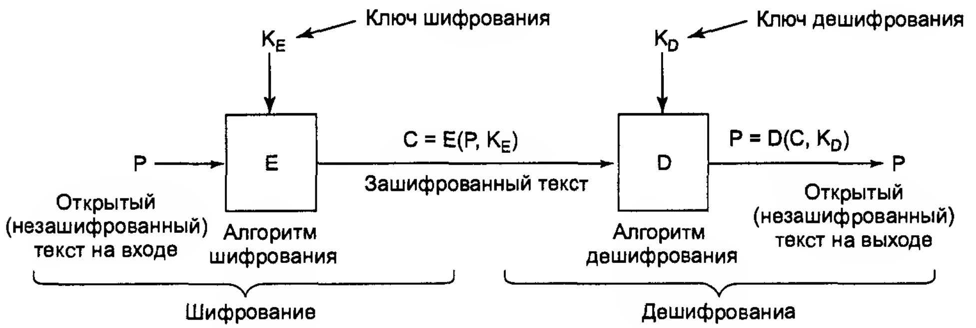


Рис. 9.1. Открытый текст и зашифрованный текст

Шифрование с секретным ключом

Рассмотрим самый простой пример алгоритма шифрования, в котором каждая буква заменяется другой буквой, например все символы A заменяются символами Q, все символы B заменяются символами W, все символы C заменяются символами E и т. д.

```
Открытый текст:         ABCDEFGHIJKLMNOPQRSTUVWXYZ
Зашифрованный текст:    QWERTYUIOPASDFGHJKLZXCVBNM
```

Такая общая схема называется **моноалфавитной подстановкой**, ключом к которой является 26-символьная строка, соответствующая полному алфавиту, то есть *QWERTYUIOPASDFGHJKLZXCVBNM*. В нашем примере слово *ATTACK* (атака) будет выглядеть как *QZZQEA*. Ключ дешифрации содержит информацию о том, как из этого слова снова получить исходный открытый текст. В данном приме-

ре ключ дешифрации представляет собой *KXVMCNOPHQRSZYIJADLEGWBUFT*, так как символу *A* в зашифрованном тексте соответствует символ *K* в открытом тексте, символу *B* в зашифрованном тексте соответствует символ *X* в открытом тексте и т. д.

На первый взгляд такая система может показаться надежной, так как даже если криптоаналитику известна общая система, он не знает, какой из $26! \approx 4\times10^{26}$ вариантов ключа применить. Тем не менее подобный шифр легко взламывается даже при довольно небольших порциях зашифрованного текста. Для подбора шифра может быть использовано преимущество статистических характеристик естественных языков. Например, в английском языке буква *e* встречается в тексте чаще всего. Следом за ней по частоте использования идут буквы *t*, *o*, *a*, *n*, *i* и т. д. Наиболее часто встречающимися комбинациями из двух символов, или **биграммами**, являются *th*, *in*, *er*, *re* и т. д. При помощи данной информации взлом такого шифра несложен.

Многие криптографические системы, как и данная система, обладают тем свойством, что по ключу шифрования легко найти ключ дешифрации, и наоборот. Такие системы называются системами **шифрования с секретным ключом** или системами **шифрования с симметричным ключом**. Хотя шифры с использованием моноалфавитной подстановки являются бесполезными, известно множество других алгоритмов с симметричным ключом, которые относительно надежны при достаточно большой длине ключа. Для серьезного уровня безопасности, вероятно, следует использовать ключи длиной в 1024 бит. При такой длине ключа пространство ключей составит $2^{1024} = 2 \times 10^{308}$ ключей. Более короткие ключи могут остановить любителей, но не специальные службы развитых государств.

Шифрование с открытым ключом

Системы с секретным ключом эффективны, так как количество вычислений для шифрования и дешифрования сообщения не очень велико, но у них есть серьезный недостаток: и отправитель, и получатель должны оба обладать общим секретным ключом. Им, возможно, даже может понадобиться физический контакт для передачи ключа. Для решения данной проблемы применяется шифрование с открытым ключом [95]. Главное свойство этой системы заключается в том, что для шифрования и дешифрования используются различные ключи и что по заданному ключу шифрования определить соответствующий ключ дешифрации практически невозможно. При таких условиях ключ шифрования может быть сделан открытым, и только ключ дешифрации будет храниться в секрете.

Чтобы дать представление о шифровании с открытым ключом, рассмотрите две следующие задачи:

Вопрос 1: Сколько будет 314159265358979 × 314159265358979?
Вопрос 2: Чему равен квадратный корень из 3912571506419387090594828508241?

Большинство шестиклассников, если дать им бумагу и карандаш, а также пообещать действительно большое сливочное мороженое с сиропом за правильный ответ, смогут ответить на первый вопрос за один-два часа. Большинство взрослых людей, получив карандаш, бумагу и обещание пожизненного 50-процентного снижения налогов, не смогут вообще решить вторую задачу без помощи калькулятора,

компьютера или еще какой-нибудь внешней помощи. Хотя возведение в квадрат и извлечение квадратного корня представляют собой взаимообратные операции, их различие в сложности является огромным. Такой вид асимметрии и формирует основу криптографии с открытым ключом. Для шифрования используется простая операция, но для дешифрации без ключа требуется выполнить огромный объем сложных вычислений.

Система шифрования с открытым ключом **RSA** использует тот факт, что перемножить два больших числа значительно легче, чем разложить большое число на множители, особенно когда используется модульная арифметика, а все большие числа состоят из сотен цифр [277]. Эта система широко используется в мире криптографии. Также применяются системы, основанные на дискретных логарифмах [106]. Главный недостаток систем шифрования с открытым ключом заключается в том, что они в тысячи раз медленнее, чем системы симметричного шифрования.

Шифрование с открытым ключом используется следующим образом. Все участники выбирают пару ключей (открытый ключ, закрытый ключ) и публикуют открытый ключ. Открытый ключ используется для шифрования, а закрытый — для дешифрации. Как правило, формирование ключей автоматизировано, иногда в качестве начального числа используется пароль, выбираемый пользователем. Чтобы отправить пользователю секретное сообщение, корреспондент зашифровывает его открытым ключом получателя. Поскольку закрытый ключ есть только у получателя, только он один сможет расшифровать сообщение.

Необратимые функции

Есть множество ситуаций, как мы увидим позднее, в которых требуется некая функция f, обладающая тем свойством, что при заданной функции f и параметре x вычисление $y = f(x)$ легко выполнимо, но по заданному $f(x)$ найти значение x невозможно по вычислениям. Такая функция, как правило, перемешивает биты сложным образом. Вначале она может присвоить числу y значение x. Затем в ней может располагаться цикл, выполняющийся столько раз, сколько в числе x содержится единичных битов. На каждом цикле биты числа y перемешиваются способом, зависящим от номера цикла, плюс к числу y прибавляются определенные константы.

Цифровые подписи

Часто бывает необходимо подписать цифровой документ цифровой подписью. Например, предположим, клиент банка посылает банку по электронной почте сообщение с поручением купить для него определенные акции. Через час после того, как это сообщение было отправлено и поручение исполнено, биржа рушится. Теперь клиент отрицает, что он отправлял сообщение банку. Банк, естественно, воспроизводит сообщение, но клиент заявляет, что банк его подделал. Как судье определить, кто говорит правду?

С помощью цифровой подписи можно подписывать сообщения, посылаемые по электронной почте, и другие цифровые документы таким образом, чтобы отправитель не смог потом отрицать, что посылал их. Один распространенный спо-

соб получения цифровой подписи заключается в том, что документ пропускается через необратимый алгоритм хэширования, который очень трудно инвертировать. Хэш-функция, как правило, формирует результат фиксированной длины, независящей от изначальной длины документа. Наиболее популярными функциями хэширования являются **MD5** (Message Digest 5 — профиль сообщения 5), создающий 16-байтовый результат [276], и алгоритм **SHA** (Secure Hash Algorithm — надежный алгоритм хэширования), формирующий 20-байтовый результат [248].

На следующем шаге предполагается использование криптографии с открытым ключом, как описывалось выше. Владелец документа с помощью своего закрытого ключа из хэша получает *D*(*hash*). Это значение, называемое **сигнатурным блоком**, добавляется к документу и посылается отправителю, как показано на рис. 9.2. Иногда применение функции *D* к хэш-коду называют дешифровкой хэша, но на самом деле эта операция не является дешифровкой, так как хэш-код не был зашифрован. Это просто математическое преобразование хэша.

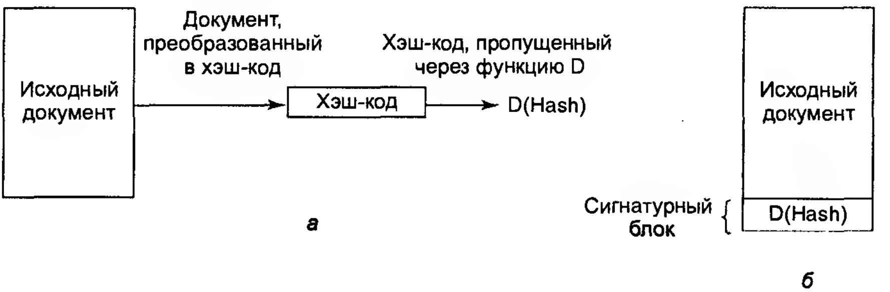


Рис. 9.2. Вычисление сигнатурного блока (*а*); что получает получатель (*б*)

Когда документ и хэш-код прибывают, получатель сначала с помощью алгоритма MD5 или SHA (о выборе алгоритма отправитель и получатель договариваются заранее) вычисляет хэш-код документа. Затем получатель применяет к сигнатурному блоку алгоритм шифрования с открытым ключом, получая *E*(*D*(*hash*)). В результате он зашифровывает «расшифрованный» хэш-код, снова получая оригинальное значение хэш-кода. Если вычисленный заново хэш-код не совпадает с расшифрованным сигнатурным блоком, это значит, что либо сообщение, либо сигнатурный блок были повреждены — или случайно, или преднамеренно. Смысл этой схемы в том, что медленное шифрование с открытым ключом применяется только для небольшого по размерам хэш-кода. Обратите внимание, что данный метод работает только в том случае, если для всех x

$$E(D(x)) = x.$$

Такое свойство не гарантировано априори для всех функций шифрования, так как все, что изначально требовалось, — это чтобы

$$D(E(x)) = x,$$

то есть E представляет собой функцию шифрования, а D — функцию дешифрования. Для возможности применения этих функций в цифровых подписях требуется, чтобы порядок применения функций не имел значения, то есть функции

D и *E* должны обладать свойством коммутативности[1]. К счастью, алгоритм RSA обладает таким свойством.

Чтобы использовать схему электронной подписи, получатель должен знать открытый ключ отправителя. Некоторые пользователи публикуют свойства открытых ключей на своих web-страницах. Другие этого не делают, так как опасаются, что злоумышленник может взломать страницу и незаметно подменить ключ. Для защиты от подобных действий требуется специальный механизм распределения открытых ключей. Один из широко применяемых методов заключается в том, что отправитель прикрепляет к сообщению **сертификат**, содержащий имя пользователя, и открытый ключ, подписанные ключом доверенной третьей стороны. Как только пользователь получит открытый ключ третьей стороны, он может получать сертификаты ото всех отправителей, использующих эту доверенную третью сторону для создания своих сертификатов.

Выше было рассказано о применении шифрования с открытым ключом для цифровых подписей. Следует отметить, что также существуют схемы, не использующие шифрования с открытым ключом.

Аутентификация пользователей

Теперь, получив некоторое представление о криптографии, перейдем к рассмотрению вопросов безопасности в операционных системах. Когда пользователь регистрируется на компьютере, операционная система, как правило, желает определить, кем является данный пользователь, и запускает процесс, называемый **аутентификацией пользователя**.

Аутентификация пользователя является одной из тех проблем, которые мы имели в виду, говоря, что «онтогенез повторяет филогенез» в соответствующем разделе главы 1. У первых мэйнфреймов, таких как ENIAC, не было операционной системы, не говоря уже о процедуре регистрации. Более поздние пакетные системы и системы разделения времени уже, как правило, имели процедуры регистрации для аутентификации заданий и пользователей.

У первых мини-компьютеров (например, PDP-1 и PDP-8) также не было процедуры регистрации, но с распространением операционной системы UNIX на миникомпьютере PDP-11 такая процедура снова потребовалась. Первые персональные компьютеры (например, Apple II и оригинальная версия IBM PC) не имели процедуры регистрации, но более сложным операционным системам для персональных компьютеров, как, например, Windows 2000, снова понадобилась процедура регистрации, обеспечивающая безопасность. Использование персонального компьютера для доступа к серверам на локальной сети или для входа на коммерческий web-сайт всегда требует регистрации. Таким образом, проблема надежной регистрации прошла несколько циклов и снова стала важной темой.

[1] Тем не менее для цифровых подписей можно использовать функции, и не обладающие подобной симметрией. Для этого нужно лишь создать новую пару ключей, из которых открытым сделать ключ дешифрации, а ключ шифрования оставить закрытым. — *Примеч. перев.*

Итак, мы определили, что аутентификация часто бывает необходима. Теперь следует найти хороший метод ее достижения. Большинство методов аутентификации пользователей основаны на распознавании:

1) чего-то, известного пользователю;
2) чего-то, имеющегося у пользователя;
3) чего-то, чем является пользователь.

На этих трех принципах построены три различные схемы аутентификации, обладающие различными сложностями и характеристиками безопасности. В следующих разделах мы рассмотрим их все по очереди.

Злоумышленнику, чтобы причинить ущерб какой-либо системе, необходимо сначала зарегистрироваться в ней. Это означает, что он должен преодолеть используемую в данной системе процедуру аутентификации. В популярной прессе таких людей называют **хакерами**[1]. Однако в компьютерном мире слово «хакер» стало чем-то вроде почетного титула, закрепляемого за великими программистами. Таким образом, термин «хакер» стал двузначным. В прессе хакерами обычно называют жуликов, однако далеко не все высококвалифицированные программисты являются мошенниками. Поэтому в данной книге, чтобы не путать эти два понятия, мы будем использовать для обозначения людей, вламывающихся в чужие компьютерные системы, слово **взломщик** (cracker).

Аутентификация с использованием паролей

В наиболее широко применяемой форме аутентификации от пользователя требуется ввести имя и пароль. Защита пароля легко реализуется. Самый простой способ реализации паролей заключается в поддержании централизованного списка пар (имя регистрации, пароль). Вводимое имя отыскивается в списке, а введенный пользователем пароль сравнивается с хранящимся в списке. Если пароли совпадают, регистрация в системе разрешается, если нет — в регистрации пользователю отказывается.

Как правило, при вводе пароля компьютер не должен отображать вводимые символы, чтобы находящиеся рядом посторонние люди не смогли узнать пароля. В операционной системе Windows 2000 при вводе каждого символа отображается звездочка. В системе UNIX при этом вообще ничего не отображается. Эти схемы обладают различными свойствами. Схема, применяемая в Windows 2000, помогает забывчивым пользователям, позволяя им видеть, сколько символов они уже набили. Однако это же свойство помогает злоумышленникам, сообщая им длину пароля. С точки зрения безопасности молчание является золотом.

Другой пример из области вопроса о правильности реализации алгоритма регистрации показан в листинге 9.1. В листинге 9.1, *а* показана успешная регистрация. Символы, вводимые пользователем, показаны как строчные, а символы, выводимые системой, — как прописные. В листинге 9.1, *б* изображена неудачная попытка взломщика войти в систему *A*. В листинге 9.1, *в* показана неудачная попытка взломщика войти в систему *B*.

[1] Изначально это слово означало плохого игрока в гольф. — *Примеч. перев.*

Листинг 9.1. Успешный вход в систему (*а*); в регистрации отказано после ввода имени (*б*); в регистрации отказано после ввода имени и пароля (*в*)

```
LOGIN: ken                    LOGIN: carol                  LOGIN: carol
PASSWORD: FooBar              INVALID LOGIN NAME            PASSWORD: Idunno
SUCCESSFUL LOGIN              LOGIN:                        INVALID LOGIN
                                                            LOGIN:
а                             б                             в
```

В листинге 9.1, *б* система отказывает в регистрации сразу, как только видит неверное имя. Такое поведение алгоритма является ошибкой, так как облегчает взломщику задачу. В этом случае он может подобрать сначала имя, не зная пароля для него. В листинге 9.1, *в* у взломщика каждый раз спрашивается и имя, и пароль, и он не знает, почему ему было отказано в регистрации, то есть было ли неверно введено имя или пароль. Все, что ему известно, — это что данная комбинация имя плюс пароль неверна.

Как взломщикам удается проникнуть в систему

Большинство взломщиков проникают в систему, просто перебирая множество комбинаций имени и пароля, пока не находят комбинацию, которая работает. Многие пользователи используют в качестве регистрационного имени свое собственное имя в той или иной форме. Например, для пользователя по имени Ellen Ann Smith разумными кандидатами регистрационного имени являются ellen, smith, ellen_smith, ellen-smith, ellen.smith, esmith, easmith и eas. Вооружившись одной из книг типа *4096 имен для вашего новорожденного*, плюс телефонной книгой, полной фамилий, взломщик без труда составит компьютеризированный список потенциальных регистрационных имен, соответствующих стране, в которой он собирается произвести атаку (имя ellen_smith может быть полезным в США или Великобритании, но вряд ли поможет в Японии).

Конечно, угадать регистрационное имя — это еще не все. Также требуется подобрать пароль. Насколько это сложно? Проще, чем вы думаете. Классический труд по вопросу безопасности паролей был написан Моррисом и Томпсоном в 1979 году на основе исследований систем UNIX [240]. Авторы данной книги скомпилировали список вероятных паролей: имя и фамилия, названия улиц, городов, слова из словарей среднего размера (также слова, написанные задом наперед), автомобильные номера и короткие строки случайных символов. Затем они сравнили свой полученный таким образом список с системным файлом паролей, чтобы посмотреть, есть ли совпадения. Как выяснилось, более 86 % от общего количества паролей в файле оказались в их списке. Аналогичный результат был получен Кляйном в 1990 году [181].

Если кто-либо полагает, что пользователи с лучшей подготовкой пользуются паролями более высокого качества, то это неверно. Исследования паролей, проведенные в финансовом районе Лондоне в 1997 году, показали, что 82 % паролей легко отгадать. Среди часто используемых паролей были сексуальные термины, ругательства, имена людей (часто членов семьи или спортивных звезд), места отдыха и различные предметы, находящиеся в офисе [169]. Это означает, что взломщик без особого труда может получить список потенциальных регистрационных имен и список потенциальных паролей.

Так ли это существенно, что пароли легко отгадать? Да. В 1998 году газета *San Jose Mercury News* сообщила, что житель городка Беркли Питер Шипли использовал несколько компьютеров в качестве устройств **автодозвона**. Эти компьютеры пытались дозвониться по всем 10 000 номерам одного коммутатора (например, (415) 770-xxxx), перебирая номера в случайном порядке, чтобы обмануть телефонные компании, пытающиеся обнаружить подобное использование компьютеров. Произведя 2,6 млн звонков, он обнаружил 20 000 компьютеров в районе залива, 200 из которых совсем не имели защиты. По оценке Питера Шипли, целеустремленный взломщик смог бы преодолеть защиту у 75 % остальных компьютеров [88].

Комбинация системы автодозвона и алгоритма подбора паролей может быть смертельной. Австралийский взломщик написал программу, систематически перебирающую все номера телефонного коммутатора, а затем пытающуюся вломиться в систему методом подбора паролей и сообщающую ему об успехе. Среди множества взломанных им систем был компьютер банка Citibank в Саудовской Аравии, что позволило ему получить номера кредитных карт и сведения о кредитном лимите (в одном случае $5 млн), а также записи транзакций (включая одно посещение публичного дома). Его коллега-взломщик также вломился в банк и собрал 4000 номеров кредитных карт [88]. Когда подобная информация используется мошенниками, банк, как правило, яростно отрицает свою вину, перекладывая всю ответственность на клиентов, которые, видимо, плохо хранили эту информацию.

Альтернативой системе автодозвона является атака на компьютеры по Интернету. У каждого компьютера в Интернете есть 32-разрядный IP-адрес, используемый для его идентификации. Люди обычно записывают эти адреса в виде четырех десятичных чисел в диапазоне от 0 до 255, разделенных точками. Взломщик легко может определить, есть ли у какого-либо компьютера некий IP-адрес и работает ли данный компьютер в настоящий момент, при помощи команды

```
ping w.x.y.z
```

Если компьютер с таким адресом включен, он ответит, и программа *ping* сообщит время прохождения сигнала в оба конца в миллисекундах (хотя некоторые сайты теперь запрещают использование программы *ping*, чтобы предотвратить подобные атаки). Нетрудно написать программу, перебирающую различные IP-адреса и подающую их на вход программе *ping*, аналогично тому, как перебирает номера телефонов обсуждавшаяся выше программа. Если по адресу *w.x.y.z* обнаружен включенный компьютер, взломщик может попытаться установить соединение с помощью команды

```
telnet w.x.y.z
```

Если попытка установки соединения принята (чего может и не быть, так как не все системные администраторы приветствуют случайную регистрацию по Интернету), взломщик может начать подбор регистрационных имен и паролей из списка[1]. Далее методом проб и ошибок взломщик может, наконец, подобрать имя и пароль, войти в систему и прочитать список паролей (расположенный в системах UNIX в файле */etc/passwd*, к которому часто разрешен доступ чтения для всех

[1] Кроме адреса компьютера, программе telnet следует еще дать на входе номер порта, который также предстоит найти перебором. — *Примеч. перев.*

пользователей). После этого взломщик может начать собирать статистику о частоте использования имен и паролей для оптимизации дальнейших поисков.

Многие демоны telnet разрывают TCP-соединение после некоторого числа неудачных попыток регистрации, чтобы затормозить процесс подбора пароля взломщиком. Взломщики отвечают на этот прием установкой множества параллельных потоков, работающих одновременно с различными компьютерами. Их цель заключается в том, чтобы выполнять столько попыток подбора пароля в секунду, сколько позволит пропускная способность линии связи. С их точки зрения вынужденное распыление сил по многим машинам не является серьезным недостатком.

Вместо того чтобы перебирать машины командой *ping* по порядку их IP-адресов, взломщик может поставить себе целью вломиться на компьютер какой-либо конкретной компании, университета или другой организации. Например, его интересует университет вымышленного города Фубар с DNS-адресом *foobar.edu*. Чтобы определить IP-адреса, используемые этим университетом, все, что ему надо сделать, — это ввести команду

```
dnsquery foobar.edu
```

и он получит список некоторых IP-адресов университета. (С той же целью может использоваться программа *nslookup* или программа *dig*.) Поскольку многие организации владеют 65 536 последовательными IP-адресами (распространенная в прошлом выделяемая область адресного пространства), узнав от программы *dnsquery* первые 2 байта IP-адреса, взломщик получает сразу информацию обо всех 65 536 адресах данной организации. Затем он может перебирать эти адреса программой *ping*, чтобы определить, какой из этих компьютеров отзывается, и попытаться установить с ним TCP-соединение программой *telnet*. Затем остается лишь подобрать имя и пароль, что мы уже обсуждали выше.

Вся последовательность действий — определение первых двух байтов IP-адреса по имени домена, перебор этих адресов программой *ping*, проверка на установку *telnet*-соединений, а затем перебор статистически вероятных пар имени и пароля — представляет собой процесс, который легко автоматизируется. Потребуется очень много попыток, чтобы взломать защиту, но если компьютеры в чем-то хороши, так это в способности без устали повторять одну и ту же последовательность операций. Взломщик с высокоскоростным кабелем или DSL-соединением (Digital Subscriber Line — цифровая абонентская линия) может запрограммировать процесс на работу в течение всего дня и просто периодически проверять, как идут дела.

Атака через Интернет, очевидно, существенно выгоднее для взломщика, чем автодозвон, так как перебор происходит значительно быстрее (не требуется время на дозванивание), а также существенно дешевле (не требуется плата за междугородние телефонные звонки). Но такой метод годится только для машин, подключенных к Интернету и принимающих *telnet*-соединения. Тем не менее многие компании (и почти все университеты) принимают *telnet*-соединения, чтобы сотрудники, находящиеся в командировках или в отдаленном офисе (или студенты из своих домов), могли получить удаленный доступ в систему.

Не только пароли пользователей часто бывают легко подбираемыми, но такое часто случается и с системными (корневыми) паролями. В частности, на многих системах после их установки не меняют пароли по умолчанию, с которыми постав-

ляется система. Клифф Столл, астроном из университета Беркли, заметил какие-то неполадки в своей системе, после чего установил ловушку против взломщика, пытавшегося проникнуть в систему [317]. Он зарегистрировал сеанс связи злоумышленника, уже вломившегося на одну машину лаборатории Lawrence Berkley (LBL) и пытавшегося проникнуть еще на одну, показанный в листинге 9.1. Учетная запись uucp (UNIX to UNIX Copy Program — протокол, используемый для обмена между согласованными UNIX-системами) используется для межмашинного сетевого трафика и обладает полномочиями суперпользователя, поэтому взломщик был на машине Министерства энергетики США в качестве суперпользователя. К счастью, лаборатория Lawrence Berkley не занимается разработкой ядерного оружия, в отличие от родственной ей лаборатории в Ливерморе. Некоторые организации полагают, что их защита лучше, но для такого мнения находится мало оснований, с тех пор как в 2000 году еще одна лаборатория ядерного оружия в Лос-Аламосе потеряла свой жесткий диск, полный секретной информации.

Листинг 9.2. Протокол сеанса связи взломщика, проникшего на компьютер Министерства энергетики США в лаборатории Lawrence Berkley

```
LBL> telnet elxsi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCDRRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Вломившись в систему и став суперпользователем, взломщик может установить **сетевой анализатор пакетов**, программу, изучающую все входящие и исходящие пакеты и пытающуюся найти в них определенные последовательности данных. Особый интерес для поиска представляют люди, регистрирующиеся с этой скомпрометированной машины на удаленных машинах, особенно с полномочиями суперпользователя на этих удаленных машинах. Эта информация может собираться для взломщика в файл, который он позднее заберет. Таким образом, взломщик, проникнув на машину со слабой защитой, часто может использовать это как средство для проникновения на другие машины с более серьезной защитой.

Последнее время все больше взломов защиты осуществляется технически безграмотными пользователями, которые просто запускают сценарии, найденные в Интернете. Эти сценарии либо применяют атаку по описанному выше методу грубой силы, либо используют известные ошибки в определенных программах. Настоящие хакеры называют таких взломщиков **script kiddies** (детишки со сценариями).

Как правило, у таких «детишек» нет конкретной цели или установки на похищение определенной информации. Они просто ищут машины, на которые легко вломиться. Некоторые используемые ими сценарии даже используют случайные сетевые адреса (старшая часть IP-адреса), после чего перебирают все машины в этой случайной сети, ища компьютер, который ответит на запрос. Когда набирается база

данных работающих IP-адресов, сценарий начинает атаку каждого компьютера по очереди. В результате может оказаться, что только что установленный секретный военный компьютер может быть атакован уже через несколько часов после подключения к Интернету, хотя никто, кроме администратора, еще не знает об этом компьютере.

Защита паролей в системе UNIX

В некоторых (старых) операционных системах файл паролей хранился на диске в незашифрованном виде, но защищался стандартными системными механизмами защиты. Хранить все пароли на диске в незашифрованном виде означает самому искать неприятности, так как многие люди слишком часто будут иметь доступ к нему. Это системные администраторы, операторы, обслуживающий персонал, программисты, руководители и, возможно, даже секретарши.

Лучшее решение выглядит следующим образом. Программа регистрации просит пользователя ввести свое имя и пароль. Пароль немедленно используется в качестве ключа для шифрации определенного блока данных. То есть выполняется необратимая функция, на вход которой подается пароль. Затем программа регистрации считывает файл паролей, представляющий собой последовательность ASCII-строк, по одной на пользователя, и находит в нем строку, содержащую имя пользователя. Если (зашифрованный) пароль, содержащийся в этой строке, совпадает с только что вычисленным зашифрованным паролем, регистрация разрешается, в противном случае в регистрации пользователю отказывается. Преимущество этой схемы состоит в том, что никто, даже суперпользователь, не может просмотреть пароли пользователей, поскольку нигде в системе они не хранятся в отрытом виде.

Однако такая схема также может быть атакована следующим образом. Взломщик сначала создает словарь, состоящий из вероятных паролей, как это делали Моррис и Томпсон. Затем они зашифровываются с помощью известного алгоритма. Не важно, сколько времени занимает данный процесс, так как все это выполняется заблаговременно. Затем, вооружившись списком пар (пароль, зашифрованный пароль), взломщик наносит удар. Он считывает (доступный для всех) файл паролей и сравнивает хранящиеся в нем зашифрованные пароли с содержимым своего списка. Каждое совпадение означает, что взломщику стали известны имя регистрации и соответствующий ему пароль. Этот процесс может быть автоматизирован простым сценарием оболочки. При запуске такого сценария, как правило, можно определить сразу несколько десятков паролей.

Осознав возможность такой атаки, Моррис и Томпсон разработали метод, делающий данный способ взлома практически бесполезным. Их идея заключается в том, что с каждым паролем связывается псевдослучайное число, состоящее из n битов, названное ими **«солью»**. При каждой смене пароля это число также меняется. Это случайное число хранится в файле в незашифрованном виде, так что все могут его читать. В файле паролей хранится не зашифрованный пароль, а зашифрованная вместе пара пароля и случайного числа. Фрагмент файла паролей показан в табл. 9.2. Для каждого из пяти пользователей в файле отведена одна строка с полями, разделенными запятыми: имя регистрации, «соль», зашифрованная пара пароль + соль. Нотация *e*(*Dog4238*) означает конкатенацию пароля пользователя Bobbie *Dog* со случайным числом *4238*, результат которой был зашифрован функцией *e*.

Таблица 9.2. Применение случайного числа для защиты от попытки взлома с помощью заранее сосчитанных зашифрованных паролей

Bobbie, 4238, e(Dog4238)
Tony, 2918, e(6%%TaeFF2918)
Laura, 6902, e(Shakespeare6902)
Mark, 1694, e(XaB@Bwcz1694)
Deborah, 1092, e(LordByron,1092)

Посмотрим теперь, как такой вариант хранения паролей повлияет на описанный выше метод взлома, при котором злоумышленник составлял список вероятных паролей, зашифровывал их и сохранял в отсортированном файле *f*, чтобы ускорить поиск пароля. Теперь, если взломщик предполагает, что *Dog* может быть паролем, ему уже недостаточно зашифровать *Dog* и поместить результат в файл *f*. Теперь ему придется зашифровать уже 2^n строк, таких как *Dog0000*, *Dog0001*, *Dog0002* и т. д., и поместить все эти варианты в файл *f*. В системе UNIX данный метод применяется с $n = 12$, поэтому размер файла *f* увеличивается в 2^{12} раз.

Для повышения надежности в некоторых современных версиях системы UNIX доступ чтения к самому файлу паролей напрямую запрещен, а для регистрации предоставляется специальная программа, просматривающая содержимое этого файла по запросу и устанавливающая задержку между запросами, чтобы существенно снизить скорость подбора паролей при любом методе. Такая комбинация добавления «соли» к паролям, запрета непосредственного чтения файла паролей и замедления доступа к файлу через специальную процедуру может противостоять многим вариантам методов взлома системы.

Совершенствование безопасности паролей

Добавление случайных чисел к файлу паролей защищает систему от взломщиков, пытающихся заранее составить большой список зашифрованных паролей, и таким образом взломать несколько паролей сразу. Однако данный метод бессилен помочь в том случае, когда пароль легко отгадать, например если пользователь *David* использует пароль *David*. Взломщик может просто попытаться отгадать пароли один за другим. Обучение пользователей в данной области может помочь, но оно редко проводится. Помимо обучения пользователей может использоваться помощь компьютера. На некоторых системах устанавливается программа, формирующая случайные, легко произносимые бессмысленные слова, такие как *fotally*, *garbungy* или *bipitty*, которые могут использоваться в качестве паролей (желательно с использованием прописных символов и специальных символов, добавленных внутрь). Программа, вызываемая пользователем для установки или смены пароля, может также выдать предупреждение при выборе слабого пароля. Среди требований программы к паролю могут быть, например, следующие:

1. Пароль должен содержать как минимум семь символов.
2. Пароль должен содержать как строчные, так и прописные символы.
3. Пароль должен содержать как минимум одну цифру или специальный символ.
4. Пароль не должен представлять собой слово, содержащееся в словаре, имя собственное и т. д.

Снисходительная программа может просто брюзжать, строгая программа может отвергать пароль и требовать ввода лучшего варианта. Программа установки пароля может также предложить свой вариант, как уже обсуждалось выше.

Некоторые операционные системы требуют от пользователей регулярной смены паролей, чтобы ограничить ущерб от утечки пароля. Недостаток такой схемы в том, что если пользователи должны слишком часто менять свои пароли, они достаточно быстро устают придумывать и запоминать хорошие пароли и переходят к простым паролям. Если система запрещает выбирать простые пароли, пользователи забывают сложные пароли и начинают записывать их на листках бумаги, приклеиваемых к мониторам, что становится главной дырой в защите.

Одноразовые пароли

Предельный вариант частой смены паролей представляет собой использование **одноразовых паролей**. При использовании одноразовых паролей пользователь получает блокнот, содержащий список паролей. Для каждого входа в систему используется следующий пароль в списке. Если взломщику и удастся узнать уже использованный пароль, он ему не пригодится, так как каждый раз используется новый пароль. Предполагается, что пользователь постарается не терять блокнот с паролями.

В действительности без такого блокнота можно обойтись, если применить элегантную схему, разработанную Лесли Лампортом, гарантирующую пользователю безопасную регистрацию в небезопасной сети при использовании одноразовых паролей [191]. Метод Лампорта может применяться для того, чтобы позволить пользователю с домашнего персонального компьютера регистрироваться на сервере по Интернету, даже в том случае, если злоумышленники смогут просматривать и копировать весь его поток в обоих направлениях. Более того, никаких секретов не нужно хранить ни на домашнем персональном компьютере, ни на сервере.

Алгоритм основан на необратимой функции, то есть функции $y = f(x)$, обладающей тем свойством, что по заданному x легко найти y, но по заданному y подобрать x невозможно по вычислениям. Вход и выход должны иметь одинаковую длину, например 128 битов.

Пользователь выбирает секретный пароль, который он запоминает. Затем он также выбирает целое число n, означающее количество одноразовых паролей, формируемое алгоритмом. Для примера рассмотрим $n = 4$, хотя на практике используются гораздо большие значения. Пусть секретный пароль равен s. Тогда первый пароль получается в результате выполнения необратимой функции $f(x)$ n раз (то есть четыре раза):

$$P_1 = f(f(f(f(s)))).$$

Второй пароль получается, если применить необратимую функцию $f(x)$ $n - 1$ раз (то есть три раза):

$$P_2 = f(f(f(s))).$$

Для получения третьего пароля нужно выполнить функцию $f(x)$ два раза, а для четвертого — один раз. Таким образом, $P_{i-1}=f(P_i)$. Основной момент, на который следует обратить здесь внимание, заключается в том, что при использовании

данного метода легко вычислить *предыдущий* пароль, но почти невозможно определить *следующий*. Например, по данному P_2 легко найти P_1, но невозможно определить P_3.

Сервер инициализируется числом P_0, представляющим собой просто $f(P_1)$. Это значение хранится в файле паролей, вместе с именем пользователя и целым числом 1, указывающим, что следующий пароль равен $f(P_1)$. Когда пользователь пытается зарегистрироваться на сервере в первый раз, он посылает на сервер свое регистрационное имя, в ответ на которое сервер высылает целое число 1, хранящееся в файле паролей. Машина пользователя отвечает числом P_1, вычисляемым локально из s, вводимого пользователем. Затем сервер вычисляет $f(P_1)$ и сравнивает результат с хранящимся в файле паролей значением P_0. Если эти значения совпадают, регистрация разрешается, целое число увеличивается на единицу, а P_1 записывается в файл поверх P_0.

При следующем входе в систему сервер посылает пользователю число 2, а машина пользователя вычисляет P_2. Затем сервер вычисляет $f(P_2)$ и сравнивает его с хранящимся в файле значением. Если эти значения совпадают, регистрация разрешается, целое число увеличивается на единицу, а P_2 записывается в файл паролей поверх P_1. И если злоумышленнику удастся узнать P_i, у него нет способа получить из него P_{i+1}, а только P_{i-1}, то есть уже использованное и теперь бесполезное значение. Когда все n паролей использованы, сервер реинициализируется новым секретным ключом.

Схема аутентификации «оклик-отзыв»

Еще один вариант идеи паролей заключается в том, что для каждого нового пользователя создается длинный список вопросов и ответов, который хранится на сервере в надежном виде (например, в зашифрованном виде). Вопросы должны выбираться так, чтобы пользователю не нужно было их записывать. Примеры возможных вопросов:

1. Как зовут сестру Марджолин?
2. На какой улице была ваша начальная школа?
3. Что преподавала миссис Воробофф?

При регистрации сервер задает один из этих вопросов, выбирая его из списка случайным образом, и проверяет ответ. Однако чтобы такая схема могла работать, потребуется большое количество пар вопросов и ответов.

Другой вариант называется **«оклик-отзыв»**. Он работает следующим образом. Пользователь выбирает алгоритм, идентифицирующий его как пользователя, например x^2. Когда пользователь входит в систему, сервер посылает ему некое случайное число, например 7. В ответ пользователь посылает серверу 49. Алгоритм может отличаться утром и вечером, в различные дни недели и т. д.

Если терминал пользователя обладает достаточной вычислительной мощностью, как, например, персональный компьютер, персональный цифровой помощник или сотовый телефон, возможно использование более мощной формы схемы «оклик-отзыв». Пользователь заранее выбирает секретный ключ k, который вручную заносится на сервер. Копия этого ключа (в зашифрованном виде) хранится на компьютере пользователя. При регистрации сервер посылает пользователю случай-

ное число r. Компьютер пользователя вычисляет и отсылает обратно значение $f(r, k)$, где f — не являющаяся секретной функция. Сервер, в свою очередь, также производит те же вычисления и сравнивает полученный результат с присланным пользователем значением. Преимущество такой схемы перед обычным паролем заключается в том, что если злоумышленник даже запишет весь трафик в обоих направлениях, он не сможет получить информацию, которая поможет ему в следующий раз. Конечно, функция f должна быть достаточно сложной, чтобы даже при большом количестве наблюдений злоумышленник не смог вычислить значение k.

Аутентификация с использованием физического объекта

Второй метод аутентификации пользователей заключается в проверке некоторого физического объекта, который есть у пользователя, а не информации, которую он знает. Например, в течение столетий применялись металлические дверные ключи. Сегодня этим физическим объектом часто является пластиковая карта, вставляемая в специальное устройство чтения, подключенное к терминалу или компьютеру. Как правило, пользователь должен не только вставить карту, но также ввести пароль, чтобы предотвратить использование потерянной или украденной карты. С этой точки зрения использование банкомата (ATM, Automatic Teller Machine) начинается с того, что пользователь регистрируется на компьютере банка с удаленного терминала (банкомата) при помощи пластиковой карты и пароля. Сегодня в большинстве стран применяется PIN-код (PIN, Personal Identification Number — личный идентификационный номер), состоящий всего из 4 цифр, что позволяет избежать необходимости установки полной клавиатуры на банкоматы.

Существует две разновидности пластиковых карт, хранящих информацию: магнитные карты и карты с процессором. Магнитные карты содержат около 140 байт информации, записанной на магнитной ленте, приклеенной к пластику. Эта информация может быть считана терминалом и передана на центральный компьютер. Часто эти данные содержат пароль пользователя (например, его PIN-код), так что терминал может сам проверить подлинность пользователя без помощи головного компьютера. Как правило, пароль шифруется ключом, известным только банку. Эти карты стоят от 10 до 50 центов, в зависимости от наличия голограммы и тиража. Применять магнитные карты для идентификации рискованно, так как устройства чтения и записи этих карт дешевы и широко распространены.

Карты, содержащие в себе микросхемы, в свою очередь, подразделяются на две категории: карты, хранящие информацию, и смарт-карты (smart card — «умная» карта). **Карты, хранящие информацию**, содержат небольшое количество памяти (как правило, менее 1 Кбайт), использующей технологию EEPROM (Electrically Erasable Programmable Read-Only Memory — электрически стираемое программируемое ПЗУ). На такой карте нет центрального процессора, поэтому сохраняемое значение должно изменяться внешним центральным процессором (в считывающем устройстве). Такие карты производятся миллионами по цене около одного доллара за штуку и широко применяются, например, в качестве телефонных карт, деньги за которые заплачены заранее. При звонке телефон просто уменьшает на единицу

значение в карте, но деньги при этом из рук в руки не переходят. По этой причине такие карты в основном выпускаются одной компанией для использования только на их машинах (скажем, телефонах или торговых автоматах). Их можно использовать для аутентификации пользователя при регистрации и хранить на них пароль размером в 1 Кбайт, который посылается на удаленный компьютер, но это редко делается.

Однако сегодня большой объем работ в сфере безопасности проводится со **смарт-картами**. На данный момент они обладают, как правило, 8-разрядным центральным процессором, работающим с тактовой частотой 4 МГц, 16 Кбайт ПЗУ, 4 Кбайт EEPROM, 512 байт временной оперативной памяти и каналом связи со скоростью 9600 бит/с для обмена данными с устройством чтения. Со временем эти «умные» карты становятся все «умнее», но они ограничены по многим параметрам, включая толщину микросхемы (так как она должна быть встроена в карту), ширину микросхемы (чтобы избежать ее поломки, когда пользователь сгибает карту) и стоимость (обычно от 5 до 50 долларов, в зависимости от мощности центрального процессора, размеров памяти и наличия или отсутствия специального криптографического сопроцессора).

Смарт-карты могут использоваться для хранения денег, как и карты, хранящие данные, но по сравнению с этим видом карт смарт-карты обладают значительно лучшими характеристиками в плане безопасности и универсальности. Эти карты можно зарядить деньгами в банкомате или дома по телефону с помощью специального устройства, поставляемого банком. Позднее, например, в магазине, пользователь может разрешить снять с карты определенную денежную сумму (напечатав YES), в результате чего карта посылает небольшое шифрованное сообщение продавцу. Затем продавец может передать это сообщение в банк, чтобы получить указанную в нем сумму.

Большое преимущество смарт-карт перед кредитными и дебетными картами состоит в том, что для использования смарт-карт не требуется соединения с банком в режиме on-line. Если вы не верите, что данное свойство является преимуществом, попытайтесь произвести следующий эксперимент. Попробуйте купить всего одну конфету в магазине и настоять на ее оплате кредитной картой. Если продавец возражает, скажите, что у вас нет с собой наличных денег, кроме того, вы стараетесь чаще расплачиваться картой, чтобы получить различные бонусы и скидки. Вы заметите, что продавец не в восторге от этой идеи (поскольку затраты, связанные с установкой соединения с банком, практически съедят всю прибыль). Смарт-карты оказываются удобными для мелких покупок, платы за телефонные разговоры, парковку автомобиля, расчетов с торговыми автоматами и многими другими устройствами, которые обычно требуют монетку. Такие устройства широко распространены в Европе и становятся все популярнее в других местах.

У смарт-карт есть много других возможных применений (хранение в надежно закодированном виде сведений о состоянии здоровья владельца, например данных об аллергии на определенные лекарства и т. п.), но нас сейчас интересует другое. Вопрос в том, как можно использовать смарт-карты для безопасной регистрации. Основная концепция проста: смарт-карта представляет собой маленький, защищенный от подделок компьютер, способный вступить в диалог (называемый **протоколом**) с центральным компьютером для аутентификации пользователя. Так,

пользователь, желающий приобрести товары на коммерческом web-сайте, может вставить смарт-карту в домашний адаптер, соединенный с персональным компьютером. Это обеспечит не только более надежную аутентификацию пользователя, но также позволит web-сайту сразу же вычесть нужную сумму из смарт-карты, что позволит избежать основных накладных расходов (и риска), связанных с использованием кредитной карты при покупках в режиме on-line.

Со смарт-картами могут применяться различные схемы аутентификации. Простой протокол «оклик-отзыв» работает следующим образом. Сервер посылает 512-разрядное случайное число смарт-карте, которая добавляет к нему 512-разрядный пароль, хранящийся в электрически стираемом программируемом ПЗУ. Затем сумма возводится в квадрат, и средние 512 бит посылаются обратно на сервер, которому известен пароль пользователя, поэтому сервер может произвести те же операции и проверить правильность результата. Эта последовательность показана на рис. 9.3. Если даже злоумышленник видит оба сообщения, он не может определить по ним пароль. Сохранять эти сообщения также нет смысла для взломщика, так как в следующий раз сервер пошлет пользователю другое 512-разрядное случайное число. Конечно, вместо возведения в квадрат может применяться (и, как правило, применяется) более хитрый алгоритм.

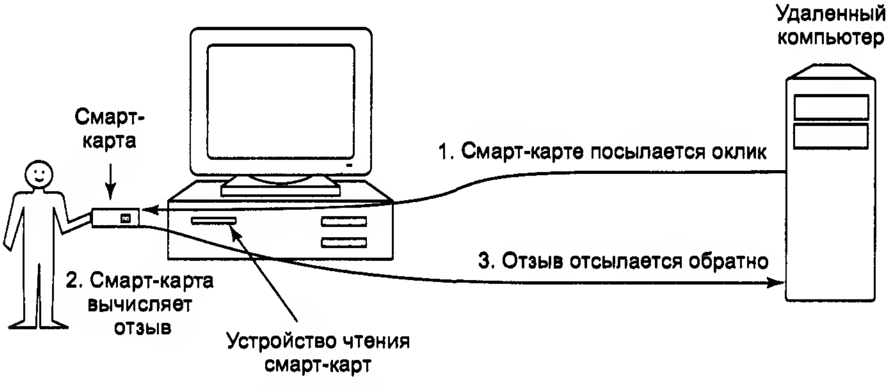


Рис. 9.3. Использование смарт-карты для аутентификации

Недостаток любого фиксированного криптографического протокола состоит в том, что со временем он может быть взломан, в результате чего смарт-карта станет бесполезной. Избежать этого можно, если хранить в памяти карты не сам криптографический протокол, а интерпретатор Java. При этом настоящий криптографический протокол будет загружаться в карту в виде двоичной программы Java и исполняться на ней. Таким образом, как только один протокол взломан, можно мгновенно перейти на использование другого протокола. Недостаток такого подхода заключается в том, что и без того не отличающаяся высокой производительностью смарт-карта будет работать еще медленнее, однако с развитием технологий этот метод приобретает все большую гибкость. Другой недостаток смарт-карт состоит в том, что потеря или кража карты может привести к раскрытию ключа при помощи анализа энергопотребления карты. Эксперт, обладающий соответствующим оборудованием, наблюдая за потребляемой картой электрической мощ-

ностью во время выполнения ею повторяемых операций шифрования, может определить ключ. Измерения времени, требуемого на зашифровку различных специально подобранных ключей, также могут дать достаточно сведений для определения секретного ключа.

Аутентификация с использованием биометрических данных

Третий метод аутентификации основан на измерении физических характеристик пользователя, которые трудно подделать. Они называются **биометрическими** параметрами [258]. Например, для идентификации пользователя может использоваться специальное устройство считывания отпечатков пальцев или тембра голоса.

Работа типичной биометрической системы состоит из двух этапов: внесение пользователя в список и идентификация. Во время первого этапа характеристики пользователя измеряются и оцифровываются. Затем извлекаются существенные особенности, которые сохраняются в записи, ассоциированной с пользователем. Эта запись может храниться на компьютере в централизованной базе данных (например, для регистрации в системе с удаленного компьютера) или в смарт-карте, которую пользователь носит с собой и вставляет в устройство чтения смарт-карт (например, банкомата).

Второй этап процесса представляет собой идентификацию. Пользователь вводит регистрационное имя. Затем система снова производит замеры. Если новые значения совпадают с хранящимися в записи, регистрация разрешается, в противном случае в регистрации пользователю отказывается. Ввод имени при регистрации нужен, так как измерения не точны, поэтому их сложно использовать в качестве индекса для поиска. Кроме того, два человека могут обладать очень близкими характеристиками, поэтому требование соответствия этих параметров для одного определенного пользователя является значительно более строгим требованием, чем простое совпадение с характеристиками любого пользователя.

Измеряемые характеристики должны отличаться у различных пользователей в достаточно широких пределах, чтобы система могла безошибочно различать разных людей. Например, цвет волос не является хорошим индикатором, так как у очень многих людей волосы одного и того же цвета. Кроме того, эти характеристики не должны сильно изменяться со временем. Например, голос человека может изменяться вследствие простуды, а лицо может выглядеть по-другому благодаря наличию или отсутствию бороды или макияжа во время начальных замеров. Поскольку последующие измерения никогда точно не совпадут с первоначальными, разработчики такой системы должны решить, насколько точным должно быть сходство. В частности, им придется решить, что лучше — отказать пару раз законному пользователю или время от времени разрешать вход в систему жулику. Коммерческий сайт может решить, что лучше терпеть небольшие убытки от мошенников, чем отказывать в регистрации постоянным клиентам, тогда как сервер лаборатории ядерного оружия может решить, что лучше иногда отказать в регистрации настоящему сотруднику, чем позволить пару раз в год войти в систему случайному чужаку.

Теперь рассмотрим некоторые биометрические показатели, использующиеся на сегодняшний день. На удивление часто на практике используется измерение дли-

ны пальцев. При этом каждый терминал оснащается устройством вроде показанного на рис. 9.4. Пользователь засовывает в него руку, и устройство измеряет длину его пальцев и сравнивается с информацией, хранящейся в базе данных.



Рис. 9.4. Устройство для измерения длины пальцев

Однако измерение длины пальцев обладает существенным недостатком. Эту систему легко обмануть, подсунув ей гипсовый (или из другого материала) слепок, возможно даже с настраиваемой длиной выдвижных пальцев, чтобы иметь возможность подбирать требуемые размеры.

Последнее время все большую популярность приобретают измерения рисунка сетчатки глаз. У каждого человека неповторимый рисунок кровеносных сосудов на сетчатке, даже у идентичных близнецов. Сетчатка глаза может быть аккуратно сфотографирована камерой с расстояния в один метр, возможно даже без ведома фотографируемого пользователя. Сетчатка глаза содержит значительно больше информации, чем отпечаток пальца. Эти данные можно закодировать, например, 256 байтами.

Подобную технику можно попытаться обмануть. Например, мошенник может подойти к банкомату, надев темные очки, к которым приклеены фотографии чужой сетчатки. В конце концов, если камера банкомата способна получить хорошую по качеству фотографию сетчатки глаза с расстояния в один метр, то кто-либо другой также сможет это сделать, и даже с больших расстояний, используя телеобъектив. По этой причине вместо фотокамер обычно используются видеокамеры, фиксирующие пульсацию, присутствующую в кровеносных сосудах сетчатки глаза.

Другой метод идентификации заключается в анализе подписи. Пользователь ставит подпись специальным пером, соединенным с терминалом, и компьютер сверяет ее с оригиналом, хранящимся на удаленном сервере или в смарт-карте. Лучше всего сравнивать не саму подпись, а движения пера и давление пера на бумагу.

Хороший специалист по подделке подписей может нарисовать довольно точную копию подписи, но он может не суметь угадать, в каком порядке выполняются движения, а также в точности повторить рисунок скорости, ускорений и давления пера.

Для измерения характеристик голоса требуется минимум специальной аппаратуры [224]. Все, что для этого нужно, — это микрофон (возможно, даже телефон); все остальное выполняется программно. В отличие от систем распознавания речи, пытающихся определить, что говорит пользователь, эти системы пытаются определить, кем является говорящий. В некоторых системах пользователь должен просто произнести скрытый пароль, но такие системы легко обманываются злоумышленниками, записывающими разговор на магнитофон и воспроизводящими его позднее. Более сложные системы говорят пользователю определенную фразу и просят повторить именно ее, причем при каждой регистрации используется новый текст. Некоторые компании начинают применять распознавание голоса в различных приложениях, таких как заказы товаров по телефону, так как подделать голос значительно сложнее, чем PIN-код.

Подобные примеры биометрических параметров можно продолжать еще и еще, но два примера помогут сделать важное замечание. Кошки и другие животные мочой метят периметр своих владений. Очевидно, кошки могут идентифицировать друг друга подобным образом. Представьте себе, что кому-нибудь удастся создать небольшое устройство, способное производить мгновенный анализ мочи, обеспечивая, таким образом, надежную идентификацию. Каждый терминал можно будет снабдить подобным устройством, снабдив терминал надписью: «Для регистрации помочитесь сюда, пожалуйста». Возможно, таким образом удалось бы создать абсолютно надежную систему, хотя она вряд ли понравилась бы пользователям.

То же самое можно сказать о системе, состоящей из иголки и небольшого спектрографа. Пользователю предлагается проколоть палец иголкой, предоставив таким образом каплю крови для спектрографического анализа. Дело в том, что любая схема аутентификации должна быть психологически приемлема для сообщества пользователей. Измерение длины пальцев, возможно, не вызовет проблем, но даже такая безобидная процедура, как снятие отпечатков пальцев, может оказаться неприемлемой, так как у многих это действие ассоциируется с обвинением в преступлении.

Контрмеры

В некоторых компьютерных системах, серьезно относящихся к вопросу безопасности (отношение к этому вопросу, как правило, кардинально меняется на следующий день после того, как злоумышленник вломился в систему и причинил серьезный ущерб), часто предпринимаются шаги, призванные усложнить несанкционированный вход в систему. Так, компания может установить политику, разрешающую регистрацию сотрудников патентного отдела только с 8 часов утра до 5 вечера с понедельника по пятницу и только с машины, находящейся в патентном отделе. Любая попытка сотрудника патентного отдела зарегистрироваться в другие часы или не с того компьютера будет расцениваться как попытка взлома системы.

Телефонные коммутируемые линии также можно сделать более безопасными. Например, всем разрешается регистрироваться в системе через модем по телефон-

ной линии, но после успешной регистрации система немедленно прерывает соединение и сама звонит пользователю по заранее условленному номеру. Такая мера означает, что взломщик не может вломиться в систему с любой телефонной линии. Для работы в системе подойдут только линии зарегистрированных пользователей. В любом случае, с применением данной техники или без нее, система должна обязательно выдерживать паузу по крайней мере в 5 с при проверке пароля и увеличивать этот временной интервал после каждой неуспешной попытки регистрации, чтоб снизить частоту попыток взломщика. После трех неуспешных попыток регистрации линия должна отключаться на 10 мин, а персонал уведомляться о попытке несанкционированного входа в систему.

Все попытки входа в систему должны регистрироваться. Когда пользователь регистрируется, система должна сообщать ему дату и время последней регистрации, а также терминал, с которого производилась эта регистрация, чтобы пользователь мог заметить взлом системы злоумышленником.

Еще один вариант защиты может заключаться в установке ловушки для взломщика. Простая схема ловушки представляет собой специальное имя регистрации с простым паролем (например, имя guest и пароль guest). При каждом входе в систему с таким именем системные специалисты в области безопасности немедленно уведомляются. Все команды, вводимые взломщиком, немедленно отображаются на мониторе руководителя службы безопасности, чтобы он мог видеть, что намеревается сделать взломщик.

Другие ловушки могут представлять собой легко обнаруживаемые ошибки в операционной системе и тому подобные вещи, намеренно разработанные с целью отлавливания злоумышленников на месте преступления. В 1989 году Столл написал занимательный доклад о ловушках, установленных им с целью поймать шпиона, вломившегося на университетский компьютер в поисках военных секретов [317].

Атаки изнутри системы

Зарегистрировавшись на компьютере, взломщик может начать причинение ущерба. Если на компьютере установлена надежная система безопасности, возможно, взломщик сможет навредить только тому пользователю, чей пароль он взломал, но часто начальная регистрация может использоваться в качестве ступеньки для последующего взлома других учетных записей. В следующих разделах будут рассмотрены разновидности атак со стороны уже зарегистрировавшегося в системе пользователя, либо взломщика, либо законного пользователя со злым умыслом против кого-либо.

Троянские кони

Одним из давно известных вариантов атаки изнутри является **троянский конь**, представляющий собой невинную с виду программу, содержащую процедуру, выполняющую неожиданные и нежелательные функции. Этими функциями могут быть удаление или шифрование файлов пользователя, копирование их туда, где их впоследствии может получить взломщик, или даже отсылка их взломщику или

во временное укромное место по электронной почте или с помощью протокола FTP. Чтобы троянский конь заработал, нужно, чтобы программа, содержащая его, была запущена. Один способ состоит в бесплатном распространении такой программы через Интернет под видом новой игры, проигрывателя MP3, «специальной» программы для просмотра порнографии и т. д., лишь бы привлечь внимание и поощрить загрузку программы. При запуске программы вызывается процедура троянского коня, которая может выполнять любые действия в пределах полномочий запустившего ее пользователя (например, удалять файлы, устанавливать сетевые соединения и т. д.). Обратите внимание, что тактика применения троянского коня позволяет обойтись без взлома компьютера жертвы.

Существуют также другие приемы, заставляющие жертву запустить троянского коня. Например, многие пользователи системы UNIX используют системную переменную окружения *$PATH*, управляющую каталогами, в которых система ищет введенную команду. Ее значение можно просмотреть следующей командой оболочки:

```
echo $PATH
```

Например, возможное значение этой переменной для пользователя *ast* может состоять из следующих каталогов:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

У других пользователей могут быть установлены другие пути поиска. Когда пользователь вводит в оболочке команду

```
prog
```

оболочка сначала ищет программу */usr/ast/bin/prog*. Если такого файла нет, оболочка пытается найти файлы */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog* и т. д., перебирая поочередно все 10 каталогов, содержащихся в строке *$PATH*. Предположим, что один из этих каталогов не был защищен, что позволило взломщику поместить туда программу. Если система, перебирая список каталогов, наткнется на эту программу, она запустится и троянский конь заработает.

Большая часть часто используемых программ хранится в каталогах */bin* или */usr/bin*, поэтому, если поместить троянского коня в файл */usr/bin/X11/ls*, то ничего не получится, так как настоящая программа будет найдена первой. Однако предположим, что взломщик поместит в каталог */usr/bin/X11* программу *la*. Если пользователь ошибочно введет *la* вместо *ls* (программы, печатающей листинг каталога), то троянский конь запустится, сделает свое грязное дело, после чего выведет сообщение, что файл *la* не найден. Таким образом, помещая троянских коней в каталоги, в которые практически никто никогда не заглядывает под именами распространенных ошибок ввода с клавиатуры, злоумышленник может рассчитывать, что рано или поздно его уловка сработает. И случайно запустить троянского коня может даже суперпользователь (даже суперпользователи иногда ошибаются при вводе команд). В этом случае троянский конь получает возможность заменить программу */bin /ls* своей версией, также содержащей троянского коня, который теперь будет запускаться при каждом обращении к программе *ls*.

Злоумышленник (назовем его Мэл), являющийся законным пользователем, может установить ловушку суперпользователю и следующим образом. Он может поместить свою версию программы *ls* в своем каталоге (скажем, *mal*), а затем сделать что-либо подозрительное, что наверняка привлечет внимание суперпользователя, например запустит одновременно 100 процессов, ограниченных производительностью процессора. Есть шанс, что суперпользователь проверит, что происходит, и введет следующую последовательность команд:

```
cd /usr/mal
ls -l
```

чтобы посмотреть, что хранится в каталоге Мэла. Поскольку некоторые оболочки сначала ищут вызываемую программу в текущем каталоге, а уже затем в каталогах, перечисленных в строке *$PATH*, суперпользователь, таким образом, вызовет троянского коня Мэла, наделив его на время запуска полномочиями суперпользователя. При этом троянский конь может, например, сделать программу */usr/mal/bin/sh* корневой с полномочиями суперпользователя. Для этого требуются всего два системных вызова: chown, чтобы сменить владельца файла */usr/mal/bin/sh* на *root*, и chmod, чтобы установить ее SETUID бит. Теперь Мэл может становиться суперпользователем, когда ему вздумается, просто запуская свой вариант оболочки.

Если Мэл часто оказывается не при деньгах, он может применить одну из следующих махинаций с троянским конем, чтобы поправить свое финансовое положение. Первый вариант заключается в том, что троянский конь проверяет, установлена ли у жертвы банковская программа, например *Quicken*, работающая в режиме on-line. Если да, то троянский конь велит этой программе перевести определенную сумму со счета жертвы на некий подставной счет (желательно в далекой стране), чтобы забрать их оттуда позднее.

Вторая жульническая схема состоит в том, что троянский конь сначала выключает звук модема, после чего звонит по платному телефонному номеру, начинающемуся с цифр 900, опять же, предпочтительно, в далекую страну, например Молдову (бывшую советскую республику). Если пользователь находился в режиме on-line во время запуска троянского коня, тогда телефонный номер в Молдове должен быть (очень дорогим) Интернет-провайдером, так что пользователь, вероятно, не заметит этого и останется в режиме on-line в течение нескольких часов. Оба приведенных варианта не являются гипотетическими. Об обоих методах мошенничества сообщалось в [88]. В последнем случае общее время телефонной связи с Молдовой составило 800 000 мин, прежде чем Федеральной торговой комиссии США удалось выйти на след и завести судебное дело против трех человек на Лонг-Айленде. В конечном итоге они согласились вернуть 38 000 жертвам 2,74 млн долларов.

Фальшивая программа регистрации

В чем-то схожа с троянскими конями жульническая схема с **фальшивой регистрацией**. Эта схема работает следующим образом. Обычно при регистрации в системе UNIX на терминале или рабочей станции, подключенной к локальной сети, пользователь видит экран, показанный на рис. 9.5, *а*. Когда пользователь садится за терминал и вводит свое регистрационное имя, система спрашивает у него пароль. Если пароль верен, пользователю разрешается вход в систему и оболочка запускается.



Рис. 9.5. Настоящий экран регистрации (*а*); фальшивый экран регистрации (*б*)

Теперь рассмотрим следующий сценарий. Мэл пишет программу, изображающую экран, показанный на рис. 9.5, *б*. Она выглядит в точности как настоящее окно, предлагающее пользователю зарегистрироваться. Теперь Мэл отходит в сторонку и наблюдает за происходящим с безопасного расстояния. Когда пользователь садится за терминал и набирает имя, программа в ответ запрашивает пароль и отключает эхо. Когда имя и пароль получены, они записываются в файл, после чего фальшивая программа регистрации посылает сигнал уничтожения собственной оболочки. В результате этого действия сеанс работы Мэла на этом терминале завершается и запускается настоящая процедура регистрации. Пользователь при этом полагает, что он неверно ввел пароль и просто регистрируется еще раз. На этот раз все проходит успешно. Но Мэлу таким образом удается получить имя и пароль. Зарегистрировавшись на нескольких терминалах и запустив на них свою обманывающую пользователей программу, он может за один день собрать много паролей.

Единственный способ защиты от подобной атаки заключается в том, чтобы программа регистрации запускалась по комбинации клавиш, которую не может перехватить программа пользователя. В системе Windows 2000 для этой цели используется комбинация клавиш CTRL+ALT+DEL. Если пользователь садится за терминал и начинает с того, что нажимает на клавиатуре CTRL+ALT+DEL, то сеанс работы текущего пользователя в системе завершается[1] и запускается системная программа регистрации. Обмануть этот механизм невозможно.

Логические бомбы

Сегодня, когда мобильность наемных работников значительно увеличилась, появилась еще одна разновидность атаки системы изнутри, называемая **логической бомбой**. Логическая бомба представляет собой программу, написанную одним из сотрудников компании и тайно установленную в операционную систему. До тех пор пока программист каждый день входит в систему под своим именем и паролем, эта программа не предпринимает никаких действий. Однако если программиста внезапно увольняют и физически удаляют из помещения без предупреждения, то на следующий день (или на следующую неделю) логическая бомба, не получив своего ежедневного пароля, начинает действовать. Существует множество вариаций на эту тему. В одном знаменитом случае программа проверяла платежную ведомость. Если личный номер программиста не появлялся в двух последовательных ведомостях подряд, бомба взрывалась [310].

[1] Точнее, в системе Windows 2000 нажатие этой комбинации клавиш, когда в системе уже зарегистрирован какой-либо пользователь, вызовет появление на экране окна менеджера системы. — *Примеч. перев.*

Взрыв логической бомбы может заключаться в форматировании жесткого диска, удалении файлов в случайном порядке, осуществлении сложно обнаруживаемых изменений в ключевых программах или шифровании важных файлов. В последнем случае компания оказывается перед сложным выбором: вызвать полицию (в результате чего много месяцев спустя злоумышленника, возможно, арестуют и признают виновным, но файлы уже не будут восстановлены) или сдаться шантажисту и снова нанять на работу этого программиста в качестве «консультанта» с астрономическим окладом для восстановления системы (и надеяться, что он при этом не заложит новые логические бомбы).

Потайные двери

Еще один способ создания дыры в системе безопасности изнутри называется **потайной дверью**. Для этого в систему системным программистом внедряется специальная программа, позволяющая обойти нормальную процедуру проверки. Например, программист может добавить к программе регистрации кусок программы, пропускающий в систему пользователя с именем «zzzzz», независимо от того, что содержится в файле паролей. Нормальный цикл процедуры регистрации может выглядеть так, как показано в листинге 9.3, *а*. В листинге 9.3, *б* показана измененная программа после установки в нее потайного входа. Процедура strcmp используется для сравнения имени регистрации со строкой «zzzzz». Если пользователь ввел при регистрации это имя, то пароль уже не важен. Если такой потайной лаз установлен программистом, работающим на фирме, производящей компьютеры, а затем поставляется вместе с компьютерами, впоследствии этот программист сможет зарегистрироваться на любом компьютере, произведенном его компанией, независимо от того, кому будет принадлежать компьютер и какая информация будет содержаться в файле паролей. Потайная дверь просто обходит весь процесс аутентификации.

Листинг 9.3. Нормальная программа (*а*); программа с установленной потайной дверью (*б*)

```
while (TRUE) {
printf("login: ");
get_string(name);
disable_echoing( );
printf("password: ");
get_string(password);
enable_echoing( );
v = check_validity(name, password);
if (v) break;
}
execute_shell(name);
```

а

```
while (TRUE) {
printf("login: ");
get_string(name);
disable_echoing( );
printf("password: ");
get_string(password);
enable_echoing( );
v = check_validity(name, password);
if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

б

Чтобы не допустить установки потайных дверей в систему, компании могут, например, ввести регулярные **просмотры программ**. При этом, когда программист закончил писать и тестировать программный модуль, модуль проверяется и помещается в базу данных. Периодически все программисты команды собираются вместе и каждый из них поочередно разъясняет остальным, что делает его программа, строка за строкой. При этом вероятность обнаружения потайных дверей значи-

тельно увеличивается. Кроме того, если программиста поймают за подобным занятием, то это не будет плюсом в его карьере. Если программисты слишком резко возражают против такой процедуры, можно поручить программистам проверку программ друг друга.

Переполнение буфера

В основе множества атак лежит тот факт, что практически все операционные системы написаны на языке программирования С (так как программисты любят этот язык, а также потому, что программы на языке С компилируются крайне эффективно). К сожалению, ни один компилятор языка С не выполняет проверки границ массива. Соответственно, следующий кусок программы представляется компилятору вполне законным:

```
int i;
char c[1024];
i = 12000;
c[i] = 0;
```

В результате выполнения этого куска программы некий байт в памяти, находящийся на 10 976 байт за пределами массива *c*, будет обнулен, возможно, с катастрофическими последствиями. Во время исполнения программы не производится никакой проверки, чтобы предотвратить эту ошибку.

Это свойство языка С позволяет произвести атаку следующего типа. На рис. 9.6, *а* показана работающая головная программа со своими локальными переменными в стеке. В некоторый момент она вызывает процедуру *A*, как показано на рис. 9.6, *б*. Стандартная процедура вызова начинается с того, что в стек помещается адрес возврата (указывающий на команду, следующую за вызовом). Затем управление передается процедуре *A*, которая помещает в стек свои локальные переменные.

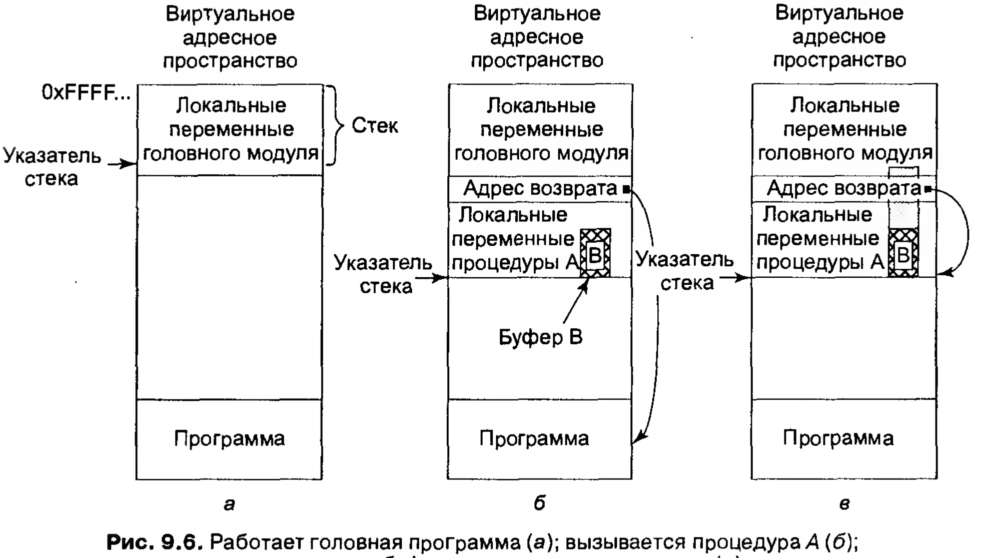


Рис. 9.6. Работает головная программа (*а*); вызывается процедура *A* (*б*); переполнение буфера показано серым цветом (*в*)

Предположим, что для работы процедуре *A* требуется получить полный путь к файлу (возможно, при помощи объединения пути текущего каталога с именем файла), после чего открыть этот файл и выполнить с ним какие-либо действия. У процедуры *A* есть буфер *B* фиксированного размера (то есть массив), в котором она содержит имя файла (рис. 9.6, *б*). Использовать буфер фиксированного размера значительно проще, чем сначала вычислить требуемый размер, а затем динамически запросить у системы буфер нужного размера. Если зарезервировать буфер длиной 1024 байт, то его должно быть достаточно для хранения любых имен файлов, не так ли? Особенно если операционная система ограничивает дину имен файлов (или, что еще лучше, длину полного пути) 255 символами.

К сожалению, в данных рассуждениях содержится фатальный просчет. Предположим, что пользователь задает этой программе имя файла длиной в 2000 символов. При этом программа не сможет открыть файл, но взломщика это не волнует. Когда процедура скопирует имя файла в свой буфер, имя файла переполнит буфер и затрет область памяти, показанную серым цветом на рис. 9.6, *в*. Что еще хуже, если имя файла достаточно длинное, оно также запишется поверх адреса возврата, поэтому, когда процедура *A* станет выполнять возврат в головной модуль, адрес возврата процедура *A* возьмет из середины имени файла. Если этот адрес представляет собой случайный мусор, то программа прыгнет по случайному адресу и, вероятно, аварийно завершится, выполнив несколько команд.

Но что, если содержимое имени файла не случайно? Что, если оно содержит работоспособную программу и тщательно настроено, так чтобы адрес возврата как раз указывал на начало этой программы, например на начало буфера *B*? При этом после возврата из процедуры *A* начнет выполняться программа в буфере *B*. Таким образом, взломщику удалось внедрить в программу свою процедуру и передать ей управление.

Тот же самый трюк может сработать с любой длинной строкой, превышающей размер зарезервированного под нее буфера. Этой строкой может быть, например, строка переменных окружения, пользовательский ввод и т. д. Предоставляя программе длинную, заранее подготовленную строку, содержащую программу, можно попасть этой программой на стек и, таким образом, добиться ее выполнения. Библиотечная функция *gets* языка С, читающая строку неизвестного размера в буфер фиксированной длины, печально известна подверженности атакам подобного рода. Некоторые компиляторы даже распознают использование функции *gets* и предупреждают об этом.

Теперь мы подходим к самой серьезной части этой истории. Предположим, что в системе UNIX атакуется программа с установленным битом SETUID, владельцем которой является root (или программа, обладающая полномочиями системного администратора в Windows 2000, что примерно то же самое). Вставленный участок программы может при помощи пары системных вызовов дать полномочия суперпользователя файлу оболочки взломщика на диске. Альтернативные варианты заключаются в использовании специально приготовленной общей библиотеки, способной причинить разнообразный ущерб, или запуска оболочки вместо текущей программы при помощи системного вызова exec, в результате чего пользователь окажется в оболочке с полномочиями суперпользователя. Значительный процент всех проблем безопасности связан с подобным дефектом программ, не

проверяющих длину данных, копируемых в буфер фиксированной длины. Исправить все программы достаточно сложно, так как системных программ, написанных на языке С и не проверяющих переполнения буфера, очень много.

Обнаружить программу, не имеющую проверки переполнения буфера, очень легко: достаточно дать ей на входе имя файла из 10 000 символов, денежную сумму в 100 цифр или что-либо подобное и посмотреть, не сломается ли она с выдачей распечатки области памяти. Затем нужно проанализировать образ памяти и определить, где располагается данная длинная строка. После этого определить, в каком месте и какие символы следует указать, чтобы перехватить управление, переписав адрес возврата, не так уж сложно. Если к тому же доступен исходный код программы, как это имеет место для большинства программ в системе UNIX, подобная атака еще более упрощается, так как содержимое стека известно заранее. Защита от атак подобного рода заключается в явной проверке длины всех поставляемых пользователем строк перед копированием этих строк в буферы. К сожалению, подверженность какой-либо программы подобной атаке, как правило, выясняется уже после успешной атаки.

Атака системы безопасности

Обычный способ проверить надежность системы безопасности состоит в приглашении группы экспертов, называемых **командой тигров**, или **группой проникновения**, чтобы посмотреть, смогут ли они взломать защиту. Иногда в качестве такой команды приглашались аспиранты [151]. За несколько лет подобные команды обнаружили множество областей, в которых операционные системы проявляют свою слабость. Ниже будут перечислены наиболее распространенные методы атак, часто завершающиеся успехом. Хотя изначально эти методы разрабатывались для атаки систем разделения времени, они часто могут применяться и для нападения на серверы локальных сетей и другие совместно используемые машины. При разработке таких систем убедитесь, что они могут выдержать атаки следующих типов:

1. Запросите страницы памяти, место на диске или магнитной ленте и просто считайте. Многие системы не очищают память при ее выделении пользователю, поэтому память и диски могут содержать много интересной информации, записанной предыдущим владельцем.
2. Попытайтесь обратиться к несуществующим системным вызовам или к имеющимся системным вызовам, но с неверными параметрами, например не того типа или слишком большой длины. Многие системы не выдерживают подобных атак.
3. Начните регистрацию, а затем нажмите клавишу `DEL`, `RUBOUT` или `BREAK` посреди процесса регистрации. В некоторых системах подобным образом удается уничтожить процесс, осуществляющий проверку пароля, и регистрация считается успешной.
4. Попытайтесь модифицировать сложные структуры операционной системы, хранящиеся в памяти пользователя (если таковые имеются). В некоторых системах (особенно на мэйнфреймах) для того, чтобы открыть файл, программа формирует большую структуру, содержащую имя файла и множе-

ство других параметров, которую передает операционной системе. При чтении и записи файла операционная система иногда сама обновляет эту структуру. Модификация некоторых полей может иметь разрушительное воздействие на безопасность.

5. Прочитайте руководство и попытайтесь найти фразы, гласящие: «Не делайте *X*». Попытайтесь проделать *X* в различных комбинациях.
6. Убедите системного администратора добавить потайную дверь с обходом важных этапов проверки безопасности для любого пользователя с вашим именем.
7. Если ничего не сработает, попытайтесь найти секретаршу системного администратора и притвориться несчастным пользователем, забывшим свой пароль. В качестве альтернативы можно попытаться подкупить секретаршу. У секретарши, как правило, есть доступ к самой разнообразной и очень интересной информации, а зарплата обычно невелика. Не следует недооценивать человеческий фактор.

Подобные и другие методы атак обсуждаются в [210]. Хотя эта статья вышла уже более четверти века тому назад, многие методы, описанные в ней, все еще работают.

Печально знаменитые дефекты системы безопасности

Как в области транспорта есть свои *Титаники*, *Конкорды* и *Гинденбурги*, у разработчиков операционных систем также есть множество вещей, о которых они предпочли бы забыть. В данном разделе мы рассмотрим некоторые интересные проблемы в области систем безопасности, имевшие место в трех операционных системах: UNIX, TENEX и OS/360.

Знаменитые дефекты системы безопасности UNIX

У утилиты *lpr* системы UNIX, печатающей файл на принтере, есть входной параметр, позволяющий удалять печатаемый файл после вывода на принтер. В ранних версиях системы UNIX любой пользователь мог распечатать и удалить с помощью этой утилиты файл паролей.

Другой способ взлома системы UNIX заключался в установке связи с файлом паролей файла с именем *core*, находящимся в рабочем каталоге пользователя. Затем взломщик намеренно вызывал сбой с сохранением образа памяти SETUID-программы, который система записывала в файл *core*, то есть поверх файла паролей. Таким образом, пользователь мог заменить содержимое файла паролей, указав в новом файле несколько строк по собственному усмотрению (задавая их в качестве аргументов команды).

Еще один метод взлома защиты системы UNIX включал использование команды

```
mkdir foo
```

Утилита *mkdir* была SETUID-программой, владельцем которой являлась система (root). Эта программа создавала i-узел для каталога *foo* при помощи системно-

го вызова mknod, после чего изменяла владельца каталога *foo* со своего идентификатора UID (то есть root) на UID пользователя. Когда система была медленной, пользователю иногда удавалось успеть быстро удалить i-узел каталога и создать связь с файлом паролей под именем *foo* после системного вызова mknod, но до системного вызова chown. Когда утилита *mkdir* выполняла системный вызов chown, она делала пользователя владельцем файла паролей. Поместив необходимые команды в файл сценария оболочки, взломщик мог пытаться выполнить эту операцию снова и снова, пока трюк не срабатывал.

Знаменитые дефекты системы безопасности TENEX

Операционная система TENEX была очень популярна на машинах DEC-10. Теперь она уже не используется, но эта система навечно останется в анналах по компьютерной безопасности благодаря следующей ошибке разработчиков. Система TENEX поддерживала страничную организацию памяти. Чтобы пользователи могли отслеживать поведение собственных программ, можно было запрограммировать систему на вызов функции пользователя при каждом страничном прерывании.

Для защиты файлов в системе TENEX также использовались пароли. Для получения доступа к файлу программа должна была указать операционной системе правильный пароль во время открытия файла. Операционная система проверяла пароль символ за символом, останавливаясь, как только видела, что пароль неверен. Чтобы взломать защиту системы TENEX, злоумышленник мог аккуратно расположить пароль в памяти так, как указано на рис. 9.7, *а*, поместив первый символ пароля в конце одной страницы, а остальные символы — в начале следующей страницы.

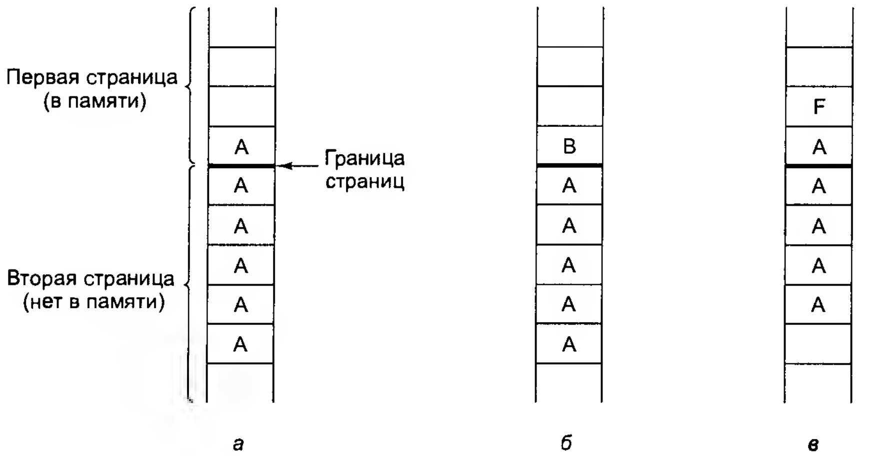


Рис. 9.7. Взлом пароля в системе TENEX

Затем необходимо гарантировать отсутствие в памяти второй страницы, для чего можно, например, много раз обратиться к другим страницам. После этого программа пыталась открыть файл жертвы с помощью тщательно подбираемого пароля. Если первый символ пароля угадан пользователем неверно, система остановит проверку на первом же символе, ответив соответствующим сообщением, и страничного прерывания не произойдет. Если же первый символ верен, операционная

система перейдет к проверке следующего символа, вызывая страничное прерывание, отслеживаемое программой пользователя.

Таким образом, программа может перебирать символ за символом, отгадывая пароль буква за буквой и сдвигая пароль при каждом отгаданном символе на один символ вниз относительно границы страниц (рис. 9.7, *в*). Для нахождения всего пароля требуется перебор менее 128 символов (набор ASCII) для каждой позиции в пароле. Следовательно, для нахождения пароля длиной в n символов требуется всего $128\,n$ попыток вместо 128^n.

Знаменитые дефекты системы безопасности OS/360

Теперь рассмотрим дефекты безопасности в системе OS/360. Описание слегка упрощено, но в нем сохранена суть проблемы. В данной системе можно было запустить чтение магнитной ленты, а затем продолжить вычисления во время считывания данных с ленты в пространство пользователя. Трюк заключался в том, чтобы начать чтение ленты, а затем обратиться к системному вызову, которому требовалась структура данных пользователя, например файл и его пароль.

Операционная система сначала проверяла верность пароля для данного файла. После этого она возвращалась и считывала имя этого файла снова уже для его чтения. (Это имя могло бы храниться в системе, но этого не делалось.) К сожалению, как раз перед тем, как операционная система собиралась считать имя файла во второй раз, поверх этого имени считывалось новое имя с магнитной ленты. Таким образом, операционная система, проверив правильность пароля для одного файла, считывала другой файл, для которого у пользователя пароля не было. Чтобы точно рассчитать время обращений к системным вызовам, требовалась определенная практика, но это было не так уж сложно.

Принципы проектирования систем безопасности

Теперь читателю должно быть ясно, что разработка хорошо защищенной операционной системы представляет собой нетривиальную задачу. Над этой проблемой без особого успеха билось множество людей. В 1975 году исследователи определили несколько общих принципов, которые необходимо использовать при проектировании надежных систем [287]. Ниже приведен краткий обзор некоторых из этих идей (основанных на опыте работы в системе MULTICS). Значимость этих идей за последние четверть века не изменилась.

Во-первых, устройство системы не должно быть секретом. Предположение, что взломщики не знают, как работает система, может только ввести разработчиков в заблуждение. Рано или поздно взломщики узнают нужную им информацию, и если защита системы окажется скомпрометированной этой утечкой информации, это значит, что такая система безопасности ни на что не годится.

Во-вторых, по умолчанию доступ не должен предоставляться. Об ошибках, в результате которых пользователям было отказано в законном доступе, сообщат значительно быстрее, чем о случаях ошибочного предоставления несанкционированного доступа. Когда есть сомнение, говорите «Нет».

В-третьих, необходимо проверять текущее состояние прав доступа. Система не должна, проверив наличие прав доступа и убедившись, что доступ разрешен, затем сохранять эту информацию для последующего использования. Многие системы

проверяют разрешение доступа при открытии файла, но не после этого. Это означает, что пользователь, открывший файл и держащий его открытым неделями, будет продолжать обладать доступом к файлу, даже если владелец файла с тех пор уже давно изменил защиту файла или, может, даже пытался удалить этот файл.

В-четвертых, предоставляйте каждому процессу как можно меньше привилегий. Если у редактора есть доступ только к редактируемому файлу (указанный при вызове редактора), редакторы, представляющие собой троянских коней с ахейцами в брюхе, не смогут причинить большого вреда. Применение этого принципа предполагает схему защиты высокой степени детализации. Подобные схемы будут рассматриваться позднее в этой главе.

В-пятых, механизм защиты должен быть простым, одинаковым для всех и встроенным в самые нижние уровни системы. Попытка установить систему безопасности на существующую незащищенную систему практически невыполнима. Безопасность, как и корректность, не является свойством, которое можно добавить потом.

В-шестых, выбранная схема должна быть психологически приемлемой. Если пользователи почувствуют, что для защиты файлов требуется затратить слишком много усилий, они не станут этого делать. Тем не менее они будут громко жаловаться, если что-либо пойдет не так. Ответы типа «это ваша вина», как правило, не будут восприниматься с пониманием.

К этому списку нам хотелось бы добавить еще один принцип, выработанный в результате десятилетий трудного опыта:

Сохраняйте схему простой.

Если система элегантна и проста, была создана одним разработчиком и имеет несколько основополагающих принципов, определяющих все остальное, то есть шанс, что она будет надежной. Если же архитектура системы представляет собой мешанину, без согласованности различных частей и со многими уступками для поддержки обратной совместимости с древними, не обладающими безопасностью системами, то в плане безопасности она будет кошмаром. Вы можете разработать систему с большим количеством свойств (функций, настроек, дружественным интерфейсом), но система с большим количеством свойств — это большая система. А большая система представляет собой потенциально небезопасную систему. Чем из большего количества строк кода состоит система, тем больше ошибок будет в системе и больше дыр будет в системе безопасности. С точки зрения безопасности, чем проще система, тем лучше.

Атаки системы снаружи

Варианты атак системы, обсуждавшиеся в предыдущих разделах, по большей части предпринимались изнутри системы, то есть уже зарегистрировавшимися пользователями. Однако последнее время, с распространением Интернета и локальных сетей, все большую угрозу для компьютеров представляют атаки снаружи. Компьютер, подключенный к сети, может быть атакован по этой сети с удаленного компьютера. Почти во всех случаях такая атака состоит из передачи по сети на атакуемую машину некоторой программы, при выполнении которой атакуемой машине наносится ущерб. По мере того как количество подключенных к Интер-

нету компьютеров продолжает увеличиваться, опасность подобных атак также растет. В следующих разделах мы рассмотрим некоторые аспекты операционных систем, связанные с внешней угрозой, уделив основное внимание вирусам, червям, мобильному коду и апплетам Java.

В последнее время сообщения об атаке компьютеров каким-либо вирусом или червем появляются в газетах почти каждый день. Вирусы и черви представляют главную проблему безопасности для отдельных пользователей и компаний. В следующих разделах мы изучим их принципы работы и обсудим, что можно предпринять для борьбы с ними.

Я долго колебался, стоит ли столь подробно описывать детали в данном разделе, не желая подвигнуть некоторых пользователей на реализацию плохих идей, но уже существующие книги содержат значительно больше подробностей, в некоторых их них даже приводятся настоящие программы [218]. Кроме того, в Интернете полно информации о вирусах, так что джин уже выпущен из бутылки. Кроме того, трудно научиться бороться с вирусами, не зная, как они работают. Наконец, у очень многих пользователей неверное представление о вирусах, которое требует коррекции.

В отличие от программистов, пишущих игры, создатели вирусов не ищут популярности после того, как их творения выходят в свет. Основываясь на скудных свидетельствах, можно предположить, что основными авторами вирусов являются старшие школьники и студенты или недавние выпускники колледжей, написавшие вирус, просто чтобы проверить, удастся ли им это, и не сознавая (или не беспокоясь), что жертв у вирусной атаки может быть столько же, сколько у урагана или землетрясения. Цель типичного автора вируса заключается в создании вируса, который быстро распространяется, который трудно обнаружить и от которого трудно избавиться после обнаружения.

Что такое вирус? В двух словах, **вирус** — это программа, которая может размножаться, присоединяя свой код к другой программе, что напоминает размножение биологических вирусов. Кроме того, вирус может выполнять и другие функции. Черви напоминают вирусов, но размножаются сами. Это отличие не будет интересовать нас в данной книге, поэтому мы будем пока использовать термин «вирус» для обоих типов программ. Черви будут рассмотрены отдельно в разделе «Интернет-черви» данной главы.

Сценарии нанесения ущерба вирусами

Поскольку вирус — это программа, он может делать то, что может программа. Например, он может выводить на экран сообщение или изображение, воспроизводить звуки или выполнять другие безвредные действия. К сожалению, он также может удалять, модифицировать, уничтожать или воровать файлы (передавая их кому-либо по электронной почте). Шантаж тоже возможен. Представьте себе вирус, который зашифровал все файлы на жестком диске жертвы, после чего вывел следующее сообщение:

```
ПРИВЕТ ОТ КОМПАНИИ GENERAL ENCRYPTION!
ДЛЯ ПРИОБРЕТЕНИЯ КЛЮЧА ДЕШИФРАЦИИ К ВАШЕМУ ЖЕСТКОМУ ДИСКУ, ПОЖАЛУЙСТА, ВЫШЛИТЕ $100 В МЕЛКИХ
НЕМАРКИРОВАННЫХ КУПЮРАХ НА А/Я 2154, ПАНАМА-СИТИ, ПАНАМА. СПАСИБО. МЫ РАДЫ СОТРУДНИЧАТЬ С ВАМИ.
```

Кроме того, вирус может сделать невозможным использование компьютера во время своей работы. Такая атака называется **атакой отказа в обслуживании**. Обычно для этого вирус поедает ресурсы компьютера, например процессорное время, или заполняет жесткий диск всяким мусором. Вот, например, программа, состоящая из одной строки, способная поставить на колени любую систему UNIX:

```
main( ) {while (1) fork( );}
```

Эта программа создает процессы, пока не переполнится таблица процессов, после чего ни один новый процесс не сможет запуститься. Теперь представьте себе вирус, заразивший в системе этим кодом каждую программу. Для защиты от подобной атаки во многих современных системах UNIX количество дочерних процессов ограничено.

Что еще хуже, вирус может повредить аппаратное обеспечение компьютера. Многие современные компьютеры содержат подсистему ввода-вывода BIOS во флэш-ПЗУ, содержимое которого может программно изменяться (чтобы проще было обновлять BIOS). Вирус может записать в BIOS случайный мусор, после чего компьютер перестанет загружаться. Если флэш-ПЗУ вставлено в кроватку, то для устранения проблемы нужно открыть компьютер и заменить микросхему. Если же флэш-ПЗУ впаяно в материнскую плату, то, возможно, всю материнскую плату придется выбросить и купить новую. Определенно невеселый опыт.

Как правило, вирусы наносят ущерб кому попало, но вирус может также преследовать и строго определенную цель. Например, компания может выпустить вирус, проверяющий, не работает ли он на компьютере конкурирующей фирмы и не отсутствует ли системный администратор в настоящий момент в системе. Если горизонт чист, он может вмешаться в производственный процесс, снижая качество продукции и создавая, таким образом, проблемы для конкурента. В остальных случаях он не будет ничего предпринимать, снижая вероятность своего обнаружения.

Другой пример вируса направленного действия — вирус, написанный амбициозным вице-президентом корпорации, который запускает его в локальную сеть собственного предприятия. Вирус проверяет, работает ли он на компьютере президента, и если да, то находит электронную таблицу и меняет в ней две случайные ячейки. Рано или поздно, основываясь на этой электронной таблице, президент примет неверное решение и будет уволен, освобождая кресло — сами понимаете для кого.

Как работает вирус

Мы достаточно налюбовались картинами разрушений. Рассмотрим теперь принципы работы вирусов. Автор вируса создает свое творение, вероятно, на ассемблере, после чего аккуратно вставляет его в программу на собственном компьютере с помощью специального инструмента, называемого **«пипеткой»** (dropper). Затем инфицированная программа распространяется, возможно, с помощью опубликования ее на BBS или в виде свободно распространяемой программы через Интернет. Эта программа может представлять собой занимательную новую игру, пиратскую версию коммерческого программного продукта или еще что-либо подобное, вызывающее интерес у публики. Затем пользователи начинают загружать эту программу на свои компьютеры.

После запуска программы вирус, как правило, начинает с того, что заражает другие программы на этой машине, после чего выполняет свою **«полезную» нагрузку**, то есть запускает ту часть программы, для которой и писался вирус. Во многих случаях эта программа может не запускаться, пока не наступит определенная дата или пока вирус гарантированно не распространится на большое число компьютеров. Выбранная дата может даже быть привязана к какому-либо политическому событию (например, к столетию или 500-летию обиды, нанесенной этнической группе автора).

Ниже мы обсудим семь разновидностей вирусов, основываясь на том, что ими заражается. Это вирусы-компаньоны, а также вирусы, заражающие исполняемые файлы, память, загрузочный сектор, драйверы устройств, макросы и исходные тексты программ. Без сомнения, вскоре появятся новые разновидности вирусов.

Вирусы-компаньоны

Вирусы-компаньоны не заражают программу, а запускаются вместо какой-либо программы. Проще всего объяснить эту концепцию на примере. В системе MS-DOS, когда пользователь вводит команду

```
prog
```

MS-DOS сначала ищет файл *prog.com*. Если такого файла нет, операционная система ищет файл *prog.exe*. В системе Windows, когда пользователь в меню Пуск выбирает пункт Выполнить..., происходит то же самое. Сегодня большинство программ представляют собой файлы с расширением *.exe*, файлы *.com* используются очень редко.

Предположим, что автору вируса известно, что многие пользователи запускают программу *prog.exe* из командной строки в MS-DOS или Windows. Он может просто создать программу, назвав ее *prog.com*. Этот файл будет запускаться каждый раз, когда кто-либо пытается запустить *prog.exe* без указания расширения файла. Выполнив свою работу, программа *prog.com* запускает файл *prog.exe*, так что пользователь ничего не замечает.

Сходный вариант атаки использует рабочий стол Windows, на котором расположены ярлыки (символьные связи) программ. Вирус может подменить путь, содержащийся в ярлыке, так, чтобы тот указывал не на программу, а на вирус. Когда пользователь щелкает дважды мышью на пиктограмме, запускается вирус. Закончив свое черное дело, вирус запускает оригинальную программу.

Вирусы, заражающие исполняемые файлы

Несколько более сложными являются вирусы, заражающие исполняемые файлы. Простейший вид таких вирусов просто записывает себя поверх исполняемой программы. Такие вирусы называются **перезаписывающими вирусами**. Логика заражения файла, содержащаяся в таком вирусе, показана в листинге 9.4.

Листинг 9.4. Рекурсивная процедура, ищущая исполняемые файлы в системе UNIX

```
#include <sys/types.h> / * стандартные заголовки POSIX * /
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
```

```
#include <unistd.h>
struct stat sbuf:                    / * для вызова lstat, чтобы убедиться, что файл
                                     / * представляет собой символьную связь * /
search(char * dir_name)
{                                    / * рекурсивный поиск исполняемых файлов * /
       DIR * dirp:                   / * указатель на открытый каталог * /
       struct dirent * dp:           / * указатель на запись каталога * /

       dirp = opendir(dir_name):     / * открыть этот каталог * /
       if (dirp == NULL) return:     / * каталог не открывается * /
       while (TRUE) {
          dp = readdir(dirp):        / * прочитать следующую запись каталога * /
          if (dp == NULL) {          / * NULL означает, что достигнут конец каталога * /
          chdir (".."):              / * вернуться в родительский каталог * /
          break:                     / * выход из цикла * /
       }
       if (dp->d_name[0] == '.') continue:     / * пропустить каталоги . и .. * /
       lstat(dp->d_name, &sbuf):               / * является ли запись символьной ссылкой? * /
       if (S_ISLNK(sbuf.st_mode)) continue:    / * пропустить символьные ссылки * /
       if (chdir(dp->d_name) == 0) {           / * если chdir завершится успешно,
                                               / * то это должен быть каталог * /
          search("."):               / * да, войти в него и продолжить поиск в нем * /
       } else {                                / * нет (файл), заразить его * /
          if (access(dp->d_name,X_OK) == 0)    / * если файл исполняемый, заразить его * /
             infect(dp->d_name):
       }
       closedir(dirp):                         / * каталог обработан: закрыть его * /
}
```

Головной модуль этого вируса сначала копирует свою двоичную программу в массив, открывая *argv*[0] и считывая его для надежного хранения. Затем он сканирует файловую систему, начиная с корневого каталога, и вызывает процедуру *search* с корневым каталогом в качестве параметра.

Рекурсивная процедура *search* обрабатывает каталог, открывая его, а затем считывая его записи по одной при помощи функции *readdir*, пока эта функция не вернет значение *NULL*. Это означает, что в каталоге больше нет записей. Если запись представляет собой каталог, он также обрабатывается рекурсивным вызовом процедуры *search*. Если же это исполняемый файл, он заражается процедурой *infect*, которой в виде параметра передается имя файла. Записи каталогов, имена которых начинаются с точки, пропускаются, чтобы избежать проблем с каталогами «.» и «..». Кроме того, пропускаются символьные связи, так как программа предполагает, что она сможет открыть каталог при помощи системного вызова chdir, а затем вернуться в предыдущий каталог, двигаясь по ссылкам «..», что возможно при использовании жестких связей, но невозможно при символьных ссылках. Для использования символьных ссылок требуется более сложная программа.

Процедура инфицирования файлов *infect* (не показанная) просто должна открыть файл по имени, записать вирус, хранящийся в массиве, в файл, после чего закрыть файл.

Этот вирус может быть «усовершенствован» различными способами. Во-первых, заражение файла может производиться не со 100-процентной вероятностью, а, скажем, лишь в одном случае из 128, для чего можно использовать генератор псевдослучайной последовательности чисел. Это позволит снизить шансы ранне-

го обнаружения вируса, в результате чего у вируса будет больше времени на распространение. Биологические вирусы обладают сходными свойствами: вирусы, которые быстро убивают свои жертвы, не распространяются так же быстро, как те, что вызывают долго длящиеся заболевания, возможно, вообще без летального исхода. В последнем случае у жертвы существенно больше шансов распространить вирус. Альтернативная схема представляет собой более высокий процент заражения (например, 25 %), но при этом есть ограничение на количество одновременно заражаемых файлов. Таким образом снижается активность диска, чтобы вызвать меньше подозрений.

Во-вторых, процедура *infect* может проверять, заражен ли уже файл. Заражение одного и того же файла дважды представляет собой просто потерю времени. В-третьих, вирус может сохранять время последнего изменения файла и размер файла неизменными, пытаясь скрыть факт заражения файла. Если программа по размеру больше, чем вирус, ее размер останется неизменным, но программы меньшие, чем вирус, увеличатся в размерах. Однако поскольку размеры большинства вирусов меньше размеров большинства программ, эта проблема не является серьезной.

Хотя эта программа не очень длинна (полный текст программы на языке C помещается на одну страницу, и размер программного сегмента не превосходит 2 Кбайт), ассемблерная версия вируса может быть еще короче. Людвиг приводит вариант написанного на ассемблере вируса для операционной системы MS-DOS, заражающего все файлы в своем каталоге и состоящего всего из 44 байт в двоичном виде [218].

Ниже в этой главе будут рассмотрены антивирусные программы, то есть программы, обнаруживающие и уничтожающие вирусы. Следует отметить, что логика, используемая вирусом для обнаружения всех исполняемых файлов (см. листинг 9.4), может применяться и в антивирусных программах. Технологии инфицирования и дезинфекции идут рука об руку, и для успешной борьбы с вирусами необходимо иметь хорошее представление о принципах работы вирусов.

С точки зрения автора вируса недостаток перезаписывающего вируса заключается в том, что его очень легко обнаружить. В конце концов, при запуске инфицированная программа сможет распространить вирус, заразив еще несколько файлов, но она не выполнит то, что должна выполнять, и пользователь это мгновенно заметит. Соответственно, большинство вирусов прицепляются к программам, позволяя им нормально выполняться после того, как вирус выполнит свое черное дело. Такие вирусы называют **паразитическими вирусами**.

Паразитические вирусы могут присоединяться в начало, конец или в середину исполняемого файла. Если вирус прикрепляется к началу файла, он должен сначала считать файл в память, записать в файл сначала себя, а затем дописать считанный файл (рис. 9.8, *б*). Однако на новом виртуальном адресе программа работать не будет, поэтому вирус либо должен настроить программу на работу по новому адресу, либо сдвинуть ее на виртуальный адрес 0 после своей работы.

Чтобы избежать сложностей, связанных с размещением вируса в начале файла, большинство вирусов прикрепляются в конец файлов и изменяют адрес запуска программы в заголовке, перенаправляя его на себя (рис. 9.8, *в*). Теперь вирус должен уметь запускаться по виртуальному адресу, зависящему от длины зараженной программы, а это означает, что вирус должен быть написан в позиционно-независимом коде, используя относительные, а не абсолютные адреса. Для опытного программиста это не сложно.

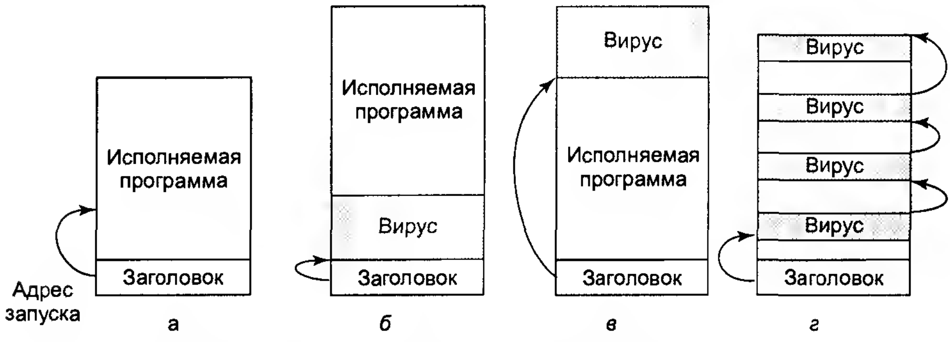


Рис. 9.8. Исполняемый файл (*а*); с вирусом в начале (*б*); с вирусом в конце (*в*); с вирусом, распределенным по свободным участкам программы (*г*)

Сложные форматы исполняемых файлов, такие как *.exe* в Windows, а также почти все современные двоичные форматы в UNIX позволяют программам состоять из нескольких сегментов текста и данных, которые загрузчик собирает в памяти и выполняет настройку адресов на лету. В некоторых системах (например, Windows) размеры всех сегментов (секций) кратны 512 байт. Если сегмент заполнен не целиком, компоновщик дополняет секцию нулями. Вирус, знакомый с этим, может попытаться спрятаться в этих промежутках. Если ему удается целиком запихать себя в свободные участки исполняемого файла, размер этого файла остается неизменным. А это является большим преимуществом, так как такой вирус сложнее обнаружить. Вирусы, использующие этот принцип, называются **полостными вирусами**. Конечно, если загрузчик не загружает пустые области в память при запуске программы, вирусу понадобится какой-либо другой способ, чтобы запуститься.

Резидентные вирусы

До сих пор мы предполагали, что при запуске зараженной программы запускается вирус, который передает управление настоящей программе, а сам завершает свое существование в памяти. В отличие от подобных вирусов **резидентные вирусы**, будучи загруженными в память, остаются там навсегда, либо прячась на самом верху памяти, либо прижимаясь «к земле» среди векторов прерываний, в которых последние несколько сот байт, как правило, не используются. Очень умные вирусы даже могут модифицировать карту памяти операционной системы, чтобы она полагала, что эта область памяти занята. Таким образом удается избежать опасности загрузки поверх вируса какой-либо другой программы.

Типичный резидентный вирус перехватывает один из векторов прерываний, сохраняя старое значение в своей переменной, и подменяет его адресом своей процедуры. Это может быть прерывание от внешнего устройства ввода-вывода или эмулированное прерывание. Лучше всего для вируса перехватывать эмулированное прерывание системного вызова. Выполнив свои дела, вирус передает управление настоящему системному вызову.

Зачем вирусу работать при каждом системном вызове? Чтобы инфицировать программы. Вирус может подождать, пока не произойдет обращение к системному вызову exec, а затем, зная, что файл, к которому происходит обращение, исполняемый (и, вероятно, полезный), заражает его. При этом процессе не требуется

значительной активности диска, как это было в случае, показанном в листинге 9.4, поэтому такой вирус имеет больше шансов остаться необнаруженным. Кроме того, перехват всех системных вызовов дает вирусу возможность шпионить за данными и выполнять самые разнообразные злодеяния.

Вирусы, поражающие загрузочный сектор

Как описывалось в главе 5, при включении большинства компьютеров BIOS считывает главную загрузочную запись с начала загрузочного диска в оперативную память и исполняет ее. Эта программа находит активный раздел диска, считывает его первый (загрузочный) сектор и исполняет его. Затем эта программа загружает либо операционную систему, либо загрузчик операционной системы. К сожалению, уже много лет назад кому-то пришла в голову идея создать вирус, перезаписывающий главную загрузочную запись или загрузочный сектор, последствия реализации которой оказались разрушительными. Такие вирусы очень широко распространены.

Как правило, вирус, поражающий загрузочный сектор (или главную загрузочную запись), сначала копирует исходное содержимое загрузочного сектора в какое-либо безопасное место на диске, что позволяет ему загружать операционную систему после того, как он закончит свои дела. Программа форматирования диска фирмы Microsoft *fdisk* пропускает первую дорожку диска, поэтому эта дорожка на машинах с операционной системой Windows представляет собой хорошее укрытие. Еще один вариант состоит в том, чтобы пометить свободный сектор диска как дефектный. Если корневой каталог достаточно велик и располагается в фиксированном месте, как в системе Windows 98, вирус может использовать конец корневого каталога. По-настоящему агрессивный вирус может даже выделить себе нормальное свободное дисковое пространство и обновить состояние списка свободных блоков соответствующим образом. Это потребует знания интимных подробностей внутренних структур данных операционной системы, но у автора вируса был хороший преподаватель на курсе операционных систем, и создатель вируса был прилежным учеником.

При загрузке компьютера вирус копирует себя в оперативную память, либо в старшие адреса, либо в неиспользуемую область векторов прерываний. В этот момент машина находится в режиме ядра, ее блок управления памятью выключен, операционной системы нет, также нет и антивирусной программы. Самый удобный момент для вирусов. Когда все готово, вирус загружает операционную систему, как правило, оставаясь резидентным в памяти.

Однако проблема таких вирусов связана с тем, как затем снова получить управление. Обычный способ заключается в использовании специфических сведений о том, как операционная система управляет вектором прерываний. Например, система Windows не перезаписывает весь вектор за одну операцию. Вместо этого она загружает драйверы устройств один за другим, и каждый из них забирает при необходимости вектор прерывания. Этот процесс может занимать около минуты.

Такая схема дает вирусу возможность сохранить за собой управление. Он начинает с того, что перехватывает сразу все векторы прерываний (рис. 9.9, *а*). По мере загрузки драйверов некоторые векторы перезаписываются. Но если только драйвер часов не загрузится первым, у вируса будет много возможностей получить управление по прерыванию от таймера. Потеря вирусом прерывания от принтера показана

на рис. 9.9, *б*. Как только вирус замечает, что один из векторов прерываний у него отнят, он может захватить его снова, зная, что теперь это безопасно. (В действительности некоторые векторы прерываний перезаписываются во время загрузки операционной системы по нескольку раз, но последовательность этих действий является детерминированной, и автор вируса знает ее наизусть.) Повторный перехват прерывания принтера показан на рис. 9.9, *в*. Когда все загружено, вирус восстанавливает все векторы прерываний, а себе оставляет только вектор эмулированного прерывания, которым пользуются системные вызовы. В конце концов, управление системными вызовами значительно интереснее, чем контроль над каждой операцией гибкого диска, но во время загрузки вирус не может рисковать потерять управление навсегда. Итак, когда операционная система загрузилась, мы имеем резидентный вирус, контролирующий системные вызовы. Большинство резидентных вирусов именно так и загружаются.

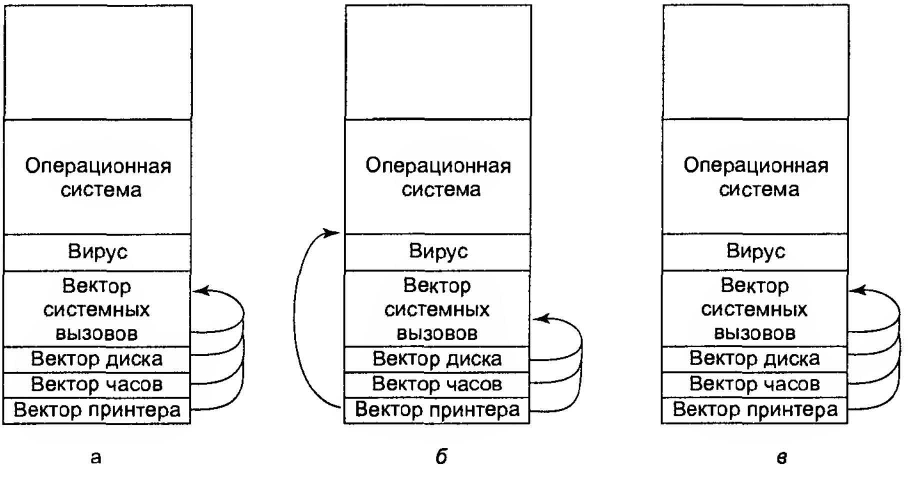


Рис. 9.9. В начале вирус перехватывает все векторы прерываний (*а*); операционная система забрала себе вектор прерывания принтера (*б*); вирус заметил потерю вектора прерывания принтера и снова захватил его (*в*)

Вирусы драйверов устройств

Описанный выше способ загрузки вируса в память напоминает спелеологию (исследование пещер) — вам нужно ползти, извиваясь в узком извилистом проходе, постоянно опасаясь, что что-то может упасть вам на голову. Было бы значительно проще, если бы операционная система была столь любезна, что грузила бы вирус законным образом. При небольших усилиях это достижимо. Трюк заключается в инфицировании драйвера устройства, чем занимаются **вирусы драйверов устройств**. В системе Windows и в некоторых UNIX-системах драйверы устройств представляют собой просто исполняемые файлы на диске, загружаемые вместе с операционной системой. Если один из них может быть заражен паразитическим вирусом, этот вирус всегда будет официально загружаться при загрузке системы. А еще при таком подходе хорошо то, что драйверы работают в режиме ядра, что позволяет вирусу перехватить вектор прерываний системных вызовов.

Макровирусы

Многие программы, такие как *Word* и *Excel*, позволяют пользователям писать макросы, чтобы группировать некоторые команды, которые позднее можно будет выполнить одним нажатием клавиши. Макросы также вызываются из меню, в которые могут добавляться новые пункты. В Microsoft *Office* макросы могут содержать целые программы на языке Visual Basic, представляющем собой полноценный язык программирования. Макросы, как правило, не компилируются, а интерпретируются, что, конечно, значительно снижает их производительность, но не ограничивает их возможностей. Поскольку обычно макросы работают с неким конкретным документом. В Microsoft *Office* макросы для каждого документа хранятся вместе с самим документом (в том же файле).

Теперь рассмотрим проблему, связанную с макросами. Автор вируса создает в редакторе *Word* документ, а также создает макрос, который присоединяет к функции OPEN FILE (открыть файл). В этом макросе содержится **макровирус**. Затем он посылает этот файл по электронной почте своей жертве, которая открывает файл (если только программа электронной почты еще не открыла этот файл сама). Открытие документа вызывает исполнение макроса OPEN FILE. Поскольку макрос может содержать произвольную программу, он может выполнять любые действия, например заражать другие документы *Word*, удалять файлы и т. д. Будем честными по отношению к корпорации Microsoft: редактор *Word* делает предупреждение при попытке открыть файл с макросами, но большинство пользователей не понимает смысла этого предупреждения и продолжает открывать файл. Кроме того, вполне безобидные документы также могут содержать макросы. И наконец, существует множество программ, открывающих файлы безо всякого предупреждения, в результате чего обнаружение вируса значительно усложняется.

С распространением электронной почты отправка документов с вирусами, встроенными в макросы, стала представлять собой постоянную головную боль. Такие вирусы написать значительно легче, чем спрятать настоящий загрузочный сектор в списке дефектных блоков, скрыть вирус среди векторов прерываний и перехватить вектор прерываний системных вызовов. Это означает, что подобные вирусы могут быть написаны значительно менее образованными людьми, снижающими качество продукции и превращающими термин «писатель вирусов» в ругательство.

Вирусы, заражающие исходные тексты программ

Паразитические вирусы и вирусы, заражающие загрузочные секторы, в высокой степени привязаны к определенной платформе. Документные вирусы в меньшей степени зависят от платформ. Самыми переносимыми вирусами являются **вирусы, заражающие исходные тексты программ**. Представьте себе вирус, показанный в листинге 9.4, но только вместо исполняемых двоичных файлов ищущий программы на языке С (для этого требуется изменить в листинге всего одну строку: обращение к процедуре access). Процедура *infect* должна вставлять в начало каждого заражаемого файла строку

```
#include <virus.h>
```

Кроме того, требуется поместить куда-либо в середину программы еще одну строку

```
run_virus( );
```

чтобы активировать вирус. Для определения места, куда вставить эту строку, от вируса требуется способность понимать структуру программы на языке C, так как это место должно синтаксически позволять вызов процедуры, к тому же в это место программы должно передаваться управление (например, бессмысленно помещать вызов процедуры после оператора return). Бесполезно также вставлять вызов процедуры run_virus() в середину комментариев. Помещение этой процедуры в цикл может оказаться тоже не очень хорошим решением. Если удастся правильно поместить вызов процедуры (например, перед самым концом процедуры *main* или перед оператором return, если такой оператор есть в тексте), то после компиляции данная программа будет содержать вирус. Вероятно файл с именем *prog.h* вызовет меньше подозрений, чем *virus.h*.

При запуске программы произойдет обращение к вирусу. Вирус может выполнять при этом самые различные действия, например искать другие программы на языке C, чтобы их заразить. Если он найдет новый файл, он может заразить его, добавив всего две приведенные выше строки, но это будет работать только на локальной машине, на которой уже установлен файл *virus.h*. Чтобы вирус мог работать на удаленной машине, необходимо включить в программу весь исходный текст вируса, который может быть вставлен в виде инициализированной текстовой строки, желательно в виде списка 32-разрядных целых чисел, чтобы было труднее догадаться, что это такое. Эта строка может быть довольно длинной, но при сегодняшних многомегабайтных программах весьма вероятно, что никто не обратит на нее внимания.

Для неподготовленного читателя все описанные выше способы заражения системы вирусами могут показаться довольно сложными. Тем не менее все эти способы применяются на практике, в чем можно убедиться, открыв любую газету. У писателей вирусов неиссякаемая фантазия, часто превосходное знание компьютера и операционных систем, а также масса свободного времени.

Как распространяются вирусы

Существует несколько сценариев распространения вирусов. Начнем наше обсуждение данной темы с классического варианта. Когда вирус создан, он помещается в какую-либо программу (как правило, чужую, хотя бывает, что автор вируса заражает им свою программу), после чего зараженная программа распространяется, например, помещается на web-сайте бесплатных или оплачиваемых после скачивания программ. Эту программу кто-нибудь скачивает и запускает. Далее может быть несколько вариантов. Во-первых, вирус может заразить несколько файлов на жестком диске в надежде, что жертва решит поделиться этими файлами со своими друзьями. Он также может попытаться заразить загрузочный сектор жесткого диска. Как только загрузочный сектор инфицирован, вирус сможет запускаться в резидентном режиме при каждой последующей загрузке компьютера.

Кроме этого, вирус может проверить наличие гибких дисков в дисководах и попытаться заразить их загрузочные секторы. Гибкие диски представляют собой удобную мишень, так как они перемещаются с машины на машину гораздо чаще, чем жесткие диски. Если загрузочный сектор гибкого диска инфицирован и такой диск используется для загрузки другого компьютера, вирус может заразить файлы и загрузочный сектор жесткого диска этого компьютера. В прошлом, когда гибкие диски представляли собой основное средство переноса программ, этот механизм был основным путем распространения вирусов.

Сегодня для распространения вирусов есть другие возможности. Вирус может проверять, подключена ли машина, на которой он работает, к локальной сети, вероятность чего очень высока для компьютеров университета или компании. Затем вирус может начать заражать незащищенные файлы на серверах этой локальной сети. На защищенные файлы эта инфекция распространиться не сможет, но часто вирусы используют специальный трюк, заключающийся в том, что намеренно вызывают странное поведение зараженных ими программ. Расчет делается на то, что пользователь, озадаченный ненормальным поведением программы, обратится за помощью к системному администратору. Системный администратор попробует сам запустить странно ведущую себя программу, чтобы посмотреть, что случилось. Если администратор выполнит это, обладая полномочиями суперпользователя, то вирус, содержащийся в программе, получит возможность заразить системные двоичные файлы, драйверы устройств, операционную систему и загрузочные секторы. Для этого потребуется всего лишь одна ошибка системного администратора, в результате которой зараженными могут оказаться все машины локальной сети.

Часто компьютеры в локальной сети имеют полномочия регистрироваться по Интернету на удаленных машинах или даже выполнять удаленно команды без регистрации. В данном случае вирусы получают еще больше возможностей для распространения. Таким образом, одна ошибка системного администратора может привести к инфицированию компьютеров всей компании. В большинстве компаний системным администраторам запрещается ошибаться.

Еще один способ распространения вирусов заключается в публикации зараженной программы в одной из конференций USENET или на BBS. Кроме того, автор вируса может создать web-страницу, для просмотра которой требуется специальный плагин (plug-in, сменный программный модуль), и тут же предложить загрузить этот плагин, который будет заражен вирусом.

В последнее время все большее распространение получают вирусы, распространяемые вместе с документами (например, редактора *Word*). Эти документы рассылаются по электронной почте или публикуются в конференциях USENET, BBS и на web-страницах Интернета, как правило, в виде файловых дополнений к письму. Даже люди, которым в голову не приходит запускать программу, присланную им незнакомым человеком, могут не понимать, что, открывая дополнение щелчком мыши, они могут впустить вирус в свою машину. Затем вирус может заглянуть в адресную книгу пользователя и разослать самого себя по всем адресам из этой книги. В строке Subject при этом, как правило, вирус указывает нечто интересное и правдоподобное, например:

```
Subject: Изменения планов
Subject: Re: то последнее письмо
```

```
Subject: Собака умерла прошлой ночью
Subject: Я серьезно болен
Subject: Я тебя люблю
```

Когда такое письмо приходит, получатель видит, что отправитель письма — его друг или коллега по работе, и ни о чем не подозревает. Когда письмо открыто, уже слишком поздно. Вирус «I LOVE YOU», распространившийся по всему миру в июне 2000 года, действовал именно этим способом и нанес ущерб в несколько миллиардов долларов.

Помимо самих вирусов, также распространяется технология их изготовления. Существуют группы писателей вирусов, активно обменивающихся информацией по Интернету и помогающих друг другу в разработке новых технологий, инструментов и вирусов. Для большинства из них создание вирусов представляет собой скорее хобби, чем профессиональную криминальную деятельность, но эффект от их действий от этого не становится менее разрушительным. Еще одну группу писателей вирусов представляют военные, рассматривающих вирусы как военное оружие, способное вывести из строя компьютерные системы противника.

С распространением вирусов связана проблема избежания обнаружения. Тюрьмы славятся плохим компьютерным оснащением, поэтому авторы вирусов предпочитают избегать этих мест. Опубликование вируса в сети со своего домашнего компьютера означает серьезный риск. Когда вирус будет, в конце концов, обнаружен, полиция сможет проследить путь его появления в сети, найдя по временному штампу самое первое сообщение, содержащее этот вирус, а по этому сообщению можно найти и его отправителя.

Чтобы минимизировать риск, автор вируса может отправить сообщение из какого-либо Интернет-кафе в другом городе. Он может принести вирус с собой на дискете и считать его самостоятельно или, если машины в Интернет-кафе не оборудованы устройствами чтения гибких дисков, попросить милую девушку за стойкой считать для него файл *book.doc*, чтобы он мог его распечатать. Получив его на свой жесткий диск, злоумышленник меняет расширение файла на *.exe* и запускает его, заражая тем самым всю локальную сеть вирусом, который срабатывает не сразу, а две недели спустя (на тот случай, если полиция решит проверить списки всех авиапассажиров, посетивших этот город за последнюю неделю). Вместо гибкого диска можно использовать удаленный FTP-сайт или принести с собой лэптоп и подключить его к Ethernet или порту USB. Подобная услуга часто предоставляется в Интернет-кафе, чтобы туристы с лэптопами могли получать свою электронную почту каждый день.

Антивирусные программы и анти-антивирусная технология

Вирусы пытаются спрятаться, а пользователи пытаются их обнаружить, играя, таким образом, друг с другом в кошки-мышки. В данном разделе мы рассмотрим некоторые вопросы на эту тему. Чтобы вируса не было видно в каталоге, вирусы, хранящие свои данные в отдельном файле (вирусы-компаньоны, вирусы исходных текстов и т. д.), должны либо устанавливать у файла бит HIDDEN (скрытый) в системе Windows, либо имя файла должно начинаться с символа точки в систе-

ме UNIX. Более сложный подход состоит в модификации проводника системы Windows или программы *ls* в UNIX, чтобы они не отображали файлы, начинающиеся с определенной последовательности символов. Вирусы также могут прятаться в необычных и неожиданных местах, таких как дефектные секторы и списки дефектных секторов, а также реестре Windows (находящейся в памяти базе данных, в которой программы могут хранить различные текстовые строки). Флэш-ПЗУ и энергонезависимая память CMOS тоже могут использоваться, хотя механизм записи во флэш-ПЗУ достаточно сложен, а CMOS-память слишком мала. И конечно, основным местом, где прячутся вирусы, остаются исполняемые файлы и документы, хранящиеся на жестком диске.

Сканеры вирусов

Очевидно, среднестатистический пользователь вряд ли способен самостоятельно обнаружить вирусы, старающиеся изо всех сил спрятаться, поэтому на рынок постоянно поступает большое количество разнообразного антивирусного программного обеспечения. Ниже мы обсудим способы работы антивирусных программ. У компаний, занимающихся разработкой антивирусных программ, есть лаборатории, в которых ученые проводят долгие часы, выслеживая и исследуя новые вирусы. Первый шаг заключается в том, чтобы заразить вирусом программу, не выполняющую никаких функций, часто называемую **козлом отпущения**, и получить вирус в его чистейшей форме. Следующий шаг состоит в том, чтобы получить точный листинг кода вируса и поместить его в базу данных известных вирусов. Компании соревнуются друг с другом, пытаясь превзойти конкурента размерами базы данных. Изобретение новых вирусов просто с целью увеличения размеров базы данных считается неспортивным.

После установки на компьютере заказчика антивирусная программа сканирует все исполняемые файлы на диске, сравнивая их содержимое с хранящимися в ее базе данных штаммами известных вирусов. У большинства компаний, занимающихся разработкой антивирусных программ, есть свои web-сайты, с которых клиенты данных компаний могут скачать описания недавно обнаруженных вирусов в свои базы данных. Если у пользователя 10 000 файлов, а в базе данных хранятся данные о 10 000 вирусах, то чтобы такая программа работала быстро, требуется очень умное программирование.

Так как незначительные мутации уже известных вирусов появляются постоянно, антивирусная программа должна уметь распознавать вирус, несмотря на изменение в трех байтах. Однако такой способ поиска не только медленнее точного поиска, но он может привести к ложным тревогам, то есть антивирусная программа будет выдавать предупреждение о незараженных файлах, которые содержат кусок кода, смутно напоминающего вирус, обнаруженный в Пакистане 7 лет назад. Пользователь при этом получает сообщение типа

```
ВНИМАНИЕ! Файл xyz.exe, возможно, заражен вирусом lahore-9x. Удалить?
```

Чем больше вирусов в базе данных и чем шире критерии поиска, тем больше будет ложных тревог. Если их будет слишком много, пользователь просто перестанет запускать антивирусную программу. Но если сканирующая программа будет искать слишком точное соответствие, она может пропустить некоторые модифи-

цированные вирусы. Найти золотую середину непросто. В идеальном случае лаборатория должна найти в вирусе некое неизменное ядро и использовать его для идентификации вируса.

Если антивирусная программа не нашла на диске вирусов на прошлой неделе, то это не означает, что их на нем нет сейчас, поэтому сканировать диск антивирусной программой следует регулярно. Поскольку процесс сканирования занимает много времени, существенно эффективнее проверять только те файлы, которые были изменены с момента последнего сканирования. Недостаток такого подхода в том, что умный вирус обязательно сохранит дату заражаемого им файла, чтобы избежать обнаружения. Антивирусная программа может проверить дату последнего изменения каталога, в котором хранится файл. Вирус может ответить на это сохранением также даты каталога. Игра в кошки-мышки продолжается.

Антивирусная программа для обнаружения изменения файлов может сохранять в своем файле на диске длины всех файлов. Если размер файла увеличился с момента последней проверки, это может означать, что он заражен, как показано на рис. 9.10, *а* и рис. 9.10, *б*. Однако умный вирус может перехитрить антивирусную программу, сжав инфицированный файл и добавив к сжатому файлу самого себя и дополнив длину файла до оригинального значения. Чтобы эта схема могла работать, вирус должен содержать процедуры сжатия и декомпрессии, как показано на рис. 9.10, *в*.

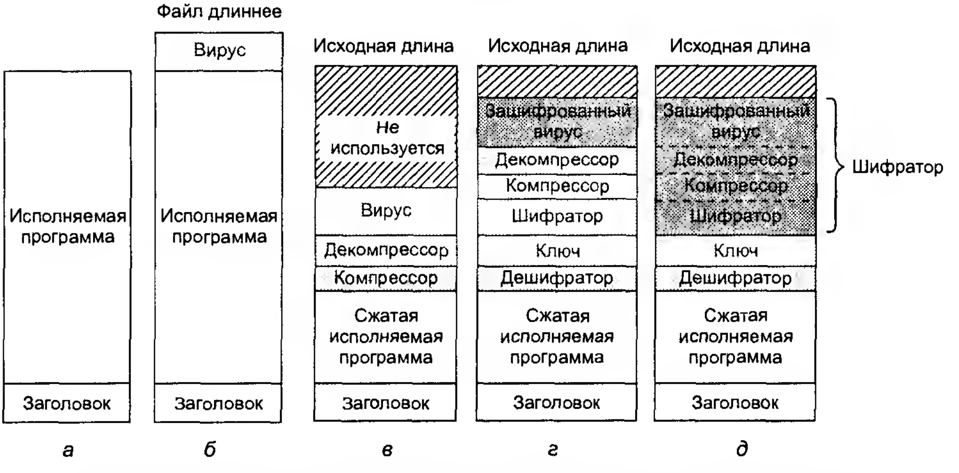


Рис. 9.10. Программа (*а*); инфицированная программа (*б*); сжатая инфицированная программа (*в*); зашифрованный вирус (*г*); сжатый вирус с зашифрованной программой компрессии (*д*)

Другой метод избежания обнаружения заключается в попытке сделать так, чтобы внешний вид вируса отличался от его представления в базе данных. Один из способов достижения этой цели заключается в том, что вирус, заражая файл, зашифровывает сам себя в этом файле, причем каждый раз используется новый ключ. Прежде чем создать новую копию, вирус формирует случайный 32-разрядный ключ шифрования, например складывая по модулю 2 текущее время с содержимым некоторых слов памяти, скажем, 72 008 и 319 992. Затем с этим ключом также по

модулю 2 складывается, слово за словом, весь код вируса. (Зашифрованное тело вируса показано темно-серым цветом на рис. 9.10, *г*.) Ключ шифрации-дешифрации хранится в самом файле. С точки зрения секретности помещение ключа в тот же файл не является идеальным решением, но цель здесь заключается в том, чтобы обмануть антивирусную программу, а не скрыть от исследователей-вирусологов детали реализации вируса. Конечно, чтобы работать, вирус должен сначала расшифровать себя, поэтому он также должен хранить в файле и процедуру дешифрации.

Эта схема все еще не совершенна, так как во всех копиях вируса будут храниться одинаковые процедуры компрессии, декомпрессии, шифрации и дешифрации, так что антивирусная программа может искать вирус по этим процедурам. Скрыть процедуры компрессии, декомпрессии и шифрации несложно: их можно зашифровать вместе с телом самого вируса (рис. 9.10, *д*). Однако процедура дешифрации сама не может быть зашифрована. Она должна быть в готовом к употреблению виде, чтобы расшифровать все остальные части вируса. Антивирусные программы это знают, и поэтому они охотятся за процедурой дешифрации.

Однако борьба на этом не заканчивается, поэтому автор вируса поступает следующим образом. Предположим, что процедуре дешифрации требуется выполнить следующие вычисления:

```
X = (A + B + C - 4)
```

Один из простейших вариантов ассемблерной программы, реализующей эту формулу для некоего двухадресного компьютера, приведен в листинге 9.5, *а*. Первый адрес в команде процессора — источник, второй — приемник, так что команда MOV A,R1 загружает содержимое переменной *A* в регистр процессора R1. Программа в листинге 9.5, *б* выполняет те же действия, но не столь эффективно, так как между полезными операциями вставлена команда NOP (No OPeration — нет операции).

Листинг 9.5. Примеры полиморфного вируса

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
		MOV R5,Y	MOV R5,Y	
а	*б*	*в*	*г*	*д*

Таким образом, существует масса способов сокрытия дешифрующей процедуры. Вместо команды NOP можно использовать самые разнообразные команды, не влияющие на ход вычислений. Например, можно прибавить 0 к какому-либо регистру, выполнить операцию логического ИЛИ для какого-либо регистра с самим собой, сдвинуть регистр влево на 0 разрядов, передать управление на следующую команду, сравнить содержимое регистров и т. д. Таким образом, программа в листинге 9.5, *в* функционально та же самая, что и в листинге 9.5, *а*. Копируя себя, вирус может использовать листинг 9.5, *в* вместо листинга 9.5, *а* и продолжать работать.

Вирус, мутирующий при каждой операции копирования, называется **полиморфным вирусом**.

Теперь предположим, что регистр R1 не используется в данном участке программы. Тогда листинг 9.5, *г* также эквивалентен листингу 9.5, *а*. Наконец, во многих случаях команды могут выполняться в другом порядке, в результате чего мы получаем еще один вариант данной процедуры (листинг 9.5, *д*). Часть вируса, занимающаяся изменением внешнего вида последовательности команд процессора, не меняя при этом их функциональности, называется **мутационным движущим механизмом.** Подобная процедура обязательно содержится в сложных вирусах, что позволяет им успешно мимикрировать. Сам мутационный механизм может быть спрятан при помощи шифрации вместе с остальными частями вируса.

Пытаться научить антивирусную программу распознавать функционально эквивалентные процедуры теоретически возможно, но практически не реально, так как это очень сильно замедлит сканирование файлов. Следует помнить, что существует очень много вариантов функционально одной и той же процедуры и если даже антивирусная программа сможет анализировать участки кода и моделировать операции с регистрами, при тысячах типов вирусов и тысячах файлов на диске у программы будет очень мало времени на тестирование, или она будет работать страшно медленно.

Отметим, что сохранение содержимого регистра R5 в ячейке *Y* было добавлено к процедуре (см. листинг 9.5, *д*), чтобы анализирующей программе было труднее определить, что все операции с этим регистром представляют собой мертвую программу, то есть они не выполняют ничего полезного. Если в других участках программы есть обращения к переменной *Y*, тогда этот кусок программы будет выглядеть вполне правдоподобным. Хорошо написанный мутационный механизм, формирующий правдоподобный и разнообразный полиморфный код, может стать кошмаром для создателей антивирусного программного обеспечения. Единственное утешение состоит в том, что подобный механизм довольно трудно написать, поэтому создатели вирусов, как правило, используют уже кем-то написанный ранее мутационный механизм. А это означает, что в обороте различных механизмов не так уж и много.

До сих пор мы обсуждали тему обнаружения вирусов в инфицированных исполняемых файлах. Помимо этого антивирусный сканер должен проверить главную загрузочную запись, загрузочные секторы, список дефектных блоков, флэш-ПЗУ, CMOS-память и т. д. Но что если в памяти находится резидентный вирус? Он не будет обнаружен. Будет еще хуже, если представить себе, что работающий резидентный вирус перехватывает все системные вызовы. Он легко сможет определить, что антивирусная программа считывает загрузочный сектор, проверяя его на вирусы. Чтобы обмануть антивирусную программу, вирусу нужно не обращаться к системному вызову. Вместо этого он считывает настоящий загрузочный сектор, хранящийся в укромном месте (например, в списке дефектных блоков). Он также завязывает себе узелок на память, чтобы не забыть снова заразить все файлы, когда антивирусный сканер завершит свою работу.

Чтобы не быть обманутой вирусом, антивирусная программа могла бы напрямую обращаться к диску, минуя системные вызовы операционной системы. Однако для этого потребуется наличие в антивирусной программе встроенных драйверов устройств для IDE, SCSI и других дисков, что снизит переносимость

антивирусной программы с машины на машину. Более того, в то время как обойти операционную систему для чтения загрузочного сектора возможно, обойтись без нее для чтения всех исполняемых файлов нельзя. Возникает опасность, что вирус сможет также обманывать антивирусную программу, подавая ей искаженные сведения об исполняемых файлах[1].

Проверка целостности

Принципиально другой метод обнаружения вирусов заключается в **проверке целостности**. Антивирусная программа, работающая подобным образом, сначала сканирует жесткий диск в поисках вирусов. Убедившись, что диск чист, она считает контрольную сумму для каждого исполняемого файла и сохранит список контрольных сумм для всех исполняемых файлов каталога в том же каталоге в файле *checksum*. При следующем запуске она пересчитывает все контрольные суммы и проверяет их соответствие данным, хранящимся в файле *checksum*. Зараженный файл будет тут же обнаружен по несовпадению контрольной суммы.

Проблема данного подхода состоит в том, что авторы вирусов не собираются сидеть сложа руки. Они могут написать вирус, удаляющий файл с контрольными суммами. Что еще хуже, можно написать вирус, считающий такую контрольную сумму заново и заменяющий старое значение в файле *checksum* новым. Чтобы защититься от подобной атаки, антивирусная программа может попытаться спрятать файл контрольных сумм, но такой подход вряд ли будет долго работать, так как авторы вирусов обязательно тщательно изучат антивирусную программу. Лучший подход заключается в шифровании файла контрольных сумм. В идеале при шифровании должна использоваться смарт-карта с ключом, хранящимся вне компьютера, чтобы программы не могли до него добраться.

Проверка поведения

Другая стратегия, используемая антивирусным программным обеспечением, состоит в **проверке поведения** программ. При этом антивирусная программа резидентно находится в памяти во время работы компьютера и сама перехватывает все системные вызовы. Идея такого подхода состоит в том, что таким образом антивирусная программа может отслеживать всю активность системы и перехватывать все, что кажется ей подозрительным. Например, ни одна нормальная программа не должна пытаться перезаписать загрузочный сектор, поэтому такие попытки почти наверняка свидетельствуют о деятельности вируса. Изменения содержимого флэш-ПЗУ тоже являются крайне подозрительными.

Но есть множество случаев не столь очевидных. Например, перезапись исполняемого файла необычна, если только это делает не компилятор. Если антивирусная программа обнаруживает подобное действие, она может издать предупреждение в расчете на то, что пользователь знает, должен ли данный файл переписываться. Если редактор *Word* перезаписывает файл с расширением *.doc* новым документом, полным макросов, это не обязательно свидетельствует об активности вируса.

[1] Эту проблему можно решить, загрузившись не с проверяемого жесткого диска, а, например, с CD-ROM или другого жесткого диска. Неясно только, что делать, если вирус заразит BIOS во флэш-ПЗУ. — *Примеч. перев.*

В системе Windows программы могут отделяться от своих исполняемых файлов и становиться резидентными при помощи специального системного вызова. Опять же, это действие может быть законным, но предупреждение стоит выдать.

Вирусы не обязательно должны спокойно сидеть и ждать, пока антивирусная программа их не убьет. Они могут сражаться. Особенно захватывающие сражения могут начаться при встрече резидентного вируса с резидентной антивирусной программой. Много лет назад программисты любили игру, называемую Core Wars (войны в памяти), в которой два программиста одновременно запускали по одной программе в общее свободное адресное пространство. Программы поочередно обращались к памяти, пытаясь определить местонахождение противника и уничтожить его прежде, чем он сделает то же самое. Сражение между вирусом и антивирусной программой выглядит похоже на эту игру, с той разницей, что местом битвы является компьютер бедного пользователя, который совсем не желает, чтобы это происходило именно на его машине. Что еще хуже, у вирусов в этой схватке есть преимущество: антивирусные программы, в отличие от вирусов, распространяются открыто, и создатели вирусов могут приобрести любую антивирусную программу для ее изучения. Конечно, как только появляется новый вирус, команды создателей антивирусных программ тут же модифицируют свои программы, вынуждая авторов вирусов добывать новые копии антивирусных программ.

Предохранение от вирусов

У каждой хорошей истории должна быть мораль. Мораль данной истории следующая:

Лучше предохраняться, чем потом сожалеть.

Постараться избежать заражения вирусом значительно проще, чем пытаться затем отыскать его на зараженном компьютере. Ниже приводится несколько советов, полезных для индивидуальных пользователей, а также обсуждаются действия, которые вся индустрия в целом может предпринять, чтобы значительно снизить остроту данной проблемы.

Что могут сделать пользователи, чтобы избежать заражения вирусом? Во-первых, выбрать операционную систему, предоставляющую определенный уровень защиты, со строгим разграничением режимов работы ядра и пользователя, а также раздельными паролями регистрации для каждого пользователя и системного администратора. При таких условиях, даже если какой-либо пользователь случайно занесет вирус в систему, этот вирус не сможет заразить системные двоичные файлы.

Во-вторых, устанавливайте только архивированное программное обеспечение, приобретенное у надежного производителя. Хотя даже это не гарантирует стопроцентного отсутствия вирусов, так как были случаи, когда рассерженные сотрудники фирмы распространяли вирусы в коммерческом программном обеспечении, тем не менее это значительно помогает. Загрузка программного обеспечения с web-сайтов и BBS весьма рискованна.

В-третьих, приобретите хорошее антивирусное программное обеспечение и используйте его так, как написано в инструкции. Обязательно получайте регулярные обновления с web-сайтов производителя.

В-четвертых, не щелкайте мышью на присоединенных к электронным письмам файлах и скажите, чтобы вам не присылали такие файлы. Электронная почта

в виде простого ASCII- текста всегда безопасна, но вложенные файлы могут быть опасны.

В-пятых, архивируйте почаще ключевые файлы на внешних носителях, таких как гибкие диски, CDR или на магнитной ленте. Храните несколько последовательных версий каждого файла на разных внешних дисках. Тогда, если вы обнаружите вирус, у вас появляется шанс восстановить файлы. Заархивированный вчера уже зараженный файл не поможет, а вот версия недельной давности может оказаться полезной.

Компьютерная промышленность также должна серьезно относиться к вирусной угрозе и изменить некоторые свои схемы поведения, представляющие опасность. Во-первых, следует производить простые операционные системы. Чем больше в операционной системе всяких прибамбасов, тем больше дыр в системе безопасности. Это медицинский факт.

Во-вторых, не применяйте активное содержимое документов. С точки зрения безопасности это катастрофа. Для просмотра присланного видеосервером документа не должна запускаться содержащаяся в документе программа. Например, JPEG-файлы не содержат программ и поэтому не могут содержать вирусов. Все документы должны работать подобным образом.

В-третьих, должен быть способ избирательно устанавливать защиту записи на цилиндры определенного диска, чтобы вирусы не могли заразить программы, хранящиеся в этих цилиндрах. Подобная защита может быть реализована с помощью битового массива в контроллере, в котором перечисляются все защищенные цилиндры. Изменение содержимого этого битового массива должно разрешаться только тогда, когда пользователь переключил механический переключатель на передней панели компьютера.

В-четвертых, флэш-ПЗУ представляет собой прекрасную идею, но запись в него также должна разрешаться только при определенном положении механического переключателя, что может сделать пользователь, устанавливая новую версию BIOS. Само собой, ни одно из этих предложений не будет восприниматься всерьез, пока действительно большой жареный вирусный петух нас всех не клюнет. Например, обнулив все банковские счета во всем мире. Хотя тогда что-либо предпринимать будет уже поздно.

Восстановление после вирусной атаки

При обнаружении вируса компьютер следует немедленно остановить, так как резидентный вирус может все еще работать. Компьютер следует перезагрузить с CD-ROM или гибкого диска, на котором всегда должна быть установлена защита от записи. На этом диске должна содержаться полная операционная система, чтобы не использовать при загрузке жесткий диск с его загрузочным сектором, копией операционной системы и драйверами, которые могут быть инфицированы. Затем с оригинального CD-ROM следует запустить антивирусную программу, так как версия программы, хранящаяся на жестком диске, также может быть заражена.

Антивирусная программа может обнаружить некоторые вирусы и может даже устранить их, но нет никакой гарантии, что она найдет их все. Вероятно, самый надежный метод заключается в том, чтобы сохранить на внешнем носителе все

файлы, которые не могут содержать вирусов (например, ASCII и JPEG-файлы). Те файлы, которые могут содержать вирусы (например, документы редактора *Word*), должны быть преобразованы в другой формат, который не может содержать вирусы, скажем, в текст ASCII (или, по крайней мере, следует удалить все макросы). Затем жесткий диск следует переформатировать программой форматирования, загруженной с защищенного от записи гибкого диска или CD-ROM, чтобы гарантировать, что сама программа форматирования не заражена вирусом. Особенно важно, чтобы главная загрузочная запись и загрузочные секторы были полностью стерты. Затем следует переустановить операционную систему с оригинального CD-ROM. Когда вы имеете дело с вирусом, не бойтесь прослыть параноиком.

Интернет-черви

Первый случай масштабного прорыва системы безопасности компьютеров, подключенных к Интернету, произошел 2 ноября 1988 года, когда аспирант университета Корнелла в штате Нью-Йорк Роберт Таппан Моррис выпустил написанного им червя в Интернет. В результате этого действия были заражены тысячи компьютеров в университетах, корпорациях и правительственных лабораториях по всему миру, прежде чем эту программу удалось выследить и удалить. Кроме того, результатом этих событий явился спор, не стихающий до сих пор. Мы обсудим ниже подробности этих событий. Дополнительные технические подробности описаны в [310]. Та же история, но в жанре полицейского триллера рассказана в [141].

История началась с того, что 1988 году Моррис обнаружил две ошибки в операционной системе Berkley UNIX, позволяющие получить несанкционированный доступ к компьютерам по всему Интернету. В одиночку он написал саморазмножающуюся программу, называемую **червем**, использующую эти ошибки и в течение секунд реплицирующую себя на каждой машине, к которой ей удается получить доступ. Он работал над этой программой несколько месяцев, тщательно настраивая ее и пытаясь научить программу заметать следы.

Неизвестно, была ли версия от 2 ноября 1988 года тестовой или окончательной. В любом случае она поставила на колени большинство систем Sun и VAX по всему Интернету за какие-то несколько часов после попадания в сеть. Мотивация Морриса не ясна, хотя он, возможно, предполагал всего лишь пошутить, но в результате программной ошибки ситуация вышла из-под контроля.

Технически червь состоял из двух программ: начального загрузчика и собственно червя. Начальный загрузчик представлял собой 99 строк на языке С (файл назывался *l1.c*). Этот загрузчик компилировался и исполнялся на атакуемом компьютере. Будучи запущенным, он связывался с машиной, с которой был загружен, загружал основного червя и запускал его. После некоторых действий, направленных на попытки скрыть свое существование, червь заглядывал в таблицы маршрутизации нового хоста, определяя компьютеры, с которыми был соединен хост, после чего пытался распространить начальный загрузчик на эти машины.

Для инфицирования новых машин применялись три метода. Метод 1 заключался в попытке запустить удаленную оболочку при помощи команды *rsh*. Некоторые компьютеры доверяют другим компьютерам и позволяют запускать *rsh*, не требуя никакой аутентификации. Если это срабатывало, удаленная оболочка загружала червя и продолжала заражать новые машины.

Метод 2 использовал программу, присутствующую на всех системах BSD, называемую *finger*. Она позволяет пользователю в Интернете ввести команду

```
finger name@site
```

чтобы отобразить информацию о владельце или администраторе конкретного компьютера. Эта информация обычно включает физическое имя, регистрационное имя, домашний адрес, телефон, имя и номер телефона секретаря, номер факса и т. д. То есть это электронный эквивалент телефонной книги.

Программа finger работает следующим образом. На каждом BSD-сайте работает фоновый процесс, называющийся **finger daemon**, отвечая на запросы, поступающие со всего Интернета. Червь обращался к программе *finger* со специально разработанной 536-байтовой строкой в качестве параметра. Эта строка вызывала переполнение буфера демона и попадала в его стек, как было показано на рис. 9.6, *в*. В данном случае использовалось отсутствие проверки на переполнение буфера. Когда процедура демона возвращала управление, она попадала не в головной модуль, а в процедуру, находящуюся внутри 536-байтовой строки в стеке. Эта процедура пыталась запустить программу *sh*. Если это ей удавалось, червь получал оболочку, работающую на атакуемой машине.

Метод 3 использовал ошибку в почтовой системе *sendmail*, позволявшей червю послать по почте копию начального загрузчика и запустить его.

Попав в систему, червь пытался взломать систему паролей. Для этого Моррису не понадобилось предпринимать собственных исследований. Все, что ему потребовалось, — это попросить у собственного отца, эксперта в области безопасности в Управлении национальной безопасности США, профессионально занимающемся взломом кодов, перепечатку классической статьи, написанной десятилетием раньше Моррисом старшим и Кеном Томпсоном в лаборатории Bell Labs [240]. Каждый взломанный пароль позволял червю зарегистрироваться на любой машине, на которой у владельца пароля были учетные записи.

Каждый раз, когда червь получал доступ к новой машине, он проверял, есть ли уже другие активные копии червя на этой машине. Если червь уже был на этой машине, то новая копия прекращала свою деятельность, кроме одного случая из семи, в котором она продолжал работать, чтобы червь мог продолжать распространяться, даже если системный администратор запустил свою версию червя, чтобы обмануть настоящего червя. Использование соотношения 1 к 7 оказалось слишком большим и привело к тому, что зараженные машины настолько заполнились червями, что просто не могли работать. Если бы Моррис не делал исключений для каждого седьмого случая, червь, возможно, до сих пор жил бы в Интернете, никем не обнаруженный.

Морриса поймали, когда один из его друзей разговаривал с репортером из компьютерной редакции *Нью-Йорк Таймс*, Джоном Марковым, и пытался убедить репортера, что все это лишь несчастный случай, что червь безобиден и автор весьма сожалеет. Друг по неосторожности упомянул, что регистрационное имя нарушителя *rtm*. Преобразовать *rtm* в физическое имя владельца было несложно — все, что Маркову надо было сделать, — это запустить процедуру *finger*. На следующий день эта история оказалась на первых полосах всех газет, вытеснив оттуда даже информацию о предстоящих через три дня президентских выборах.

Моррис предстал перед федеральным судом и был приговорен к штрафу в размере 10 000 долларов, трем годам испытательного срока и 400 часам общественных работ. Его судебные издержки, вероятно, превысили 150 000 долларов. Приговор породил множество разногласий. В компьютерном обществе многие считали, что Моррис был блестящим студентом, чья опасная шалость вышла из-под контроля. Написанная им программа не содержала ничего, что бы свидетельствовало о намерениях Морриса украсть какие-либо данные или причинить какой-либо намеренный ущерб. Другие считали его серьезным преступником, которому место в тюрьме.

Одним из результатов этого инцидента было создание группы компьютерной «скорой помощи» **CERT** (Computer Emergency Rresponse Team), основными задачами которой являются доклады о попытках взлома в Интернете, а также анализ проблем безопасности и разработка методов их решения. При необходимости эта группа рассылает свою информацию тысячам системных администраторов по Интернету. К сожалению, этой информацией, содержащей сообщения об ошибках в системах, могут воспользоваться также и злоумышленники (возможно, притворяющиеся системными администраторами) и использовать часы (или даже дни), требующиеся на устранение обнаруженных ошибок.

Мобильные программы

Вирусы и черви представляют собой программы, попадающие на компьютер без ведома владельца компьютера и вопреки его желанию. Однако иногда пользователи импортируют и запускают чужие программы на своем компьютере более или менее намеренно. Как правило, это происходит следующим образом. В далеком прошлом (что в мире Интернета означает в прошлом году) большинство web-страниц представляли собой статичные HTML-файлы с несколькими связанными с ними изображениями. Сегодня все большее число web-страниц содержат программы, называемые **апплетами**. Когда загружается web-страница, содержащая апплеты, апплеты скачиваются на компьютер и выполняются. Например, апплет может содержать форму, которую следует заполнить, с интерактивной справочной системой, помогающей при ее заполнении. Когда форма заполнена, она может быть отправлена куда-либо по Интернету на обработку. Эта технология может применяться для заполнения налоговых деклараций, заказа товаров и т. д.

Другой пример программ, переносимых с одной машины на другую для выполнения на удаленной машине, представляют собой **агенты**. Это программы, запускаемые пользователем для выполнения определенной задачи и сообщения результата. Например, агента можно попросить найти на web-сайтах путешествий самый дешевый рейс от Амстердама до Сан-Франциско. Прибывая на каждый сайт, агент запускается на нем, собирает требуемую информацию и перемещается на следующий web-сайт. Выполнив свою работу, он может вернуться и доложить о собранных им сведениях.

Третий пример мобильной программы — это файл в формате PostScript, который должен быть распечатан на принтере, поддерживающем этот формат. Файл в формате PostScript в действительности представляет собой программу на языке программирования PostScript, исполняющуюся внутри принтера. Обычно эта программа велит принтеру начертить определенные кривые линии, а затем заполнить

их каким-либо цветом, но она может также выполнять и другие действия. Апплеты, агенты и файлы в формате PostScript — это всего лишь три примера мобильных программ, однако существует еще множество других разновидностей мобильных программ.

С учетом приведенного выше долгого обсуждения вирусов и червей должно быть ясно, что позволять чужой программе работать на вашей машине несколько рискованно. Тем не менее многим хотелось бы иметь возможность запускать эти чужие программы. Так возникает вопрос: «Можно ли запускать мобильные программы безопасно?» Краткий ответ на этот вопрос звучит так: «Да, но это нелегко». Фундаментальная проблема заключается в том, что процесс импортирует апплет или другую мобильную программу в свое адресное пространство и исполняет ее. Импортированная программа работает как часть пользовательского процесса и обладает всеми полномочиями, которыми обладает пользователь, включая возможность читать, писать, удалять или зашифровывать файлы пользователя на жестком диске, посылать данные по электронной почте в дальние страны и многое другое.

Давным-давно в операционных системах была разработана концепция процессов, чтобы отгородить пользователей друг от друга. Идея заключается в том, что у каждого пользователя есть свое адресное пространство и свой идентификатор UID, что позволяет ему получать доступ к файлам и другим ресурсам, принадлежащим ему, но не другим пользователям. Защиты ресурсов и одной части процесса от другой части процесса (апплета) концепция процесса предоставить не может. Концепция потоков обеспечивает возможность параллельного исполнения нескольких задач в одном адресном пространстве процесса, но не предоставляет никакой защиты одного потока от другого.

Теоретически запуск каждого апплета в виде отдельного процесса может несколько помочь, но часто оказывается невыполнимым. Например, web-страница может содержать два или более апплетов, взаимодействующих друг с другом, а также с данными web-страницы. Web-браузеру может также понадобиться взаимодействовать с апплетами, запуская и останавливая их, снабжая их данными и т. д. Если каждый апплет поместить в свой собственный процесс, все это не сможет работать. Более того, помещение каждого апплета в свое собственное адресное пространство нисколько не усложняет апплету кражу информации или причинение ущерба.

Были предложены и реализованы новые разнообразные методы работы с апплетами (и мобильными программами в целом). Ниже мы рассмотрим три из этих методов: «песочницы», интерпретацию и программы с подписями. У каждого метода есть свои достоинства и недостатки.

«Песочницы»

Первый метод, метод «песочниц», представляет собой попытку установить для апплета во время исполнения ограниченный диапазон виртуальных адресов [349]. Суть метода состоит в том, что виртуальное адресное пространство делится на области равного размера, называемые «песочницами». Обязательным свойством каждой песочницы является одинаковость старших разрядов адресов в пределах одной песочницы. Так, 32-разрядное адресное пространство может быть разделено на 256 песочниц, разделенных границами адресов, кратных 16 Мбайт; таким образом,

у всех адресов в пределах одной песочницы будут одинаковые верхние 8 бит. С тем же успехом данное адресное пространство можно разделить на 512 песочниц по 8 Мбайт, в этом случае каждая песочница получает 9-разрядный адресный префикс. Размер песочницы следует выбирать так, чтобы в нее помещался самый большой апплет, но при этом так, чтобы не тратить на апплеты слишком много виртуального адресного пространства. Физическая память не является проблемой, если замещение страниц по требованию присутствует, а оно, как правило, присутствует. Каждому апплету выделяется две песочницы, одна для программы и одна для данных, как показано на рис. 9.11 для случая 16 песочниц по 16 Мбайт.



Рис. 9.11. Память, разделенная на 16 песочниц по 16 Мбайт

Основная идея песочниц заключается в гарантировании, что апплет не сможет передать управление программе за пределами песочницы или обратиться к данным вне ее. Две песочницы выделяются для того, чтобы апплет не смог модифицировать свою собственную программу во время исполнения, обходя данные ограничения. Запретом изменения песочницы, содержащей программу, устраняется опасность самомодифицирующихся программ. Пока апплет ограничивается подобным образом, он не может причинить ущерб браузеру или другим апплетам, установить вирусы в памяти или еще каким-либо способом повредить память.

Как только апплет загружен, он настраивается на работу по предоставляемым ему адресам песочниц. Затем выполняется проверка ограниченности всех адресных ссылок пределами соответствующей песочницы. Ниже мы будем рассматривать только ссылки на программу (то есть команды перехода JMP и CALL), но то же самое справедливо для обращений к данным. Статические команды JMP, использующие прямую адресацию, проверить легко. Не намного сложнее проверить команды передачи управления, использующие относительную адресацию. Если обнаруживается, что апплет собирается предпринять попытки выскочить за пределы своей песочницы, он просто отвергается и не запускается. Аналогично проверяются обращения к данным.

Труднее проверить команды, использующие косвенную адресацию, например динамические команды JMP. У большинства машин есть команда, передающая управление по адресу, содержащемуся в каком-либо регистре, значение которого вычисляется во время исполнения программы, например JMP (R1). Допустимость такой команды можно проверить во время исполнения. Для этого непосредственно перед выполнением этой команды прямо в код программы помещается несколько команд, сравнивающих содержимое этого регистра с границами песочницы. Пример такой проверки приведен в листинге 9.6. Как вы помните, у всех допустимых адресов должны быть одинаковые старшие *k* бит. Префикс можно хранить в регистре, например, S2. Этот регистр не должен использоваться апплетом, для чего, возможно, потребуется переписать весь апплет[1].

Листинг 9.6. Один из способов проверки допустимости команды

```
MOV R1, S1
SHR #24, S1
CMP S1, S2
TRAPNE
JMP (R1)
```

Программа работает следующим образом. Сначала исследуемый адрес копируется во вспомогательный регистр S1. Затем этот регистр сдвигается вправо, в результате чего в регистре получается значение префикса песочницы. Затем вычисленное таким образом значение сравнивается с эталоном, хранящимся в регистре S2. Если они не совпадают, происходит эмулированное прерывание и апплет уничтожается. Для проверки требуется четыре команды и два вспомогательных регистра.

Для установки подобных вставок в двоичную программу во время ее выполнения требуются определенные усилия, но ничего невыполнимого в этом нет. Было бы значительно проще, если бы апплеты поступали в виде исходных текстов, а затем компилировались локально доверенным компилятором, автоматически проверяющим статические адреса и вставляющим необходимые команды для проверки динамических адресов. В том и другом случае накладные расходы по проверке динамических адресов составляют всего около 4 %, что вполне приемлемо [349].

Вторая проблема заключается в системных вызовах, к которым пытается обращаться апплет. Решается эта проблема следующим образом. Каждый системный вызов в апплете заменяется вызовом специального модуля, называемого **монитором обращений**. Делается это на том же проходе, на котором вставляются макросы проверки динамических адресов (или если доступен исходный текст, то при компоновке апплета может использоваться специальная библиотека, обращающаяся к монитору обращений, вместо системных вызовов). В любом случае монитор обращений изучает каждую попытку системного вызова и решает, безопасно ли его выполнение. Если известно, что обращение к данному системному вызову может быть опасно или же монитор обращений в обратном не уверен, апплет уничтожается. Если монитор обращений способен определить, какой апплет к нему обратился, то может использоваться один общий монитор, обслуживающий запросы ото всех апплетов. Разрешения монитору обращений, как правило, задаются в файле конфигурации.

---

[1] Чем выделять на это целый регистр, да еще переписывать апплет, гораздо проще сравнивать регистр S1 с константой. — *Примеч. перев.*

Интерпретация

Второй способ выполнения апплетов, которым нельзя доверять, заключается в том, что апплету не передается управление, а вместо этого его шаг за шагом выполняет интерпретатор. Этот метод применяется web-браузерами. Апплеты web-страниц обычно пишутся на языке Java, представляющем собой нормальный язык программирования, либо на высокоуровневом языке сценариев, например safe-TCL или Javascript. Апплеты Java сначала компилируются в виртуальный машинный стек-ориентированный язык, называемый **JVM**[1] (Java Virtual Machine — виртуальная машина Java). Именно эти JVM-апплеты помещаются в web-страницы. После загрузки они помещаются в JVM-интерпретатор, содержащийся в браузере, как показано на рис. 9.12.

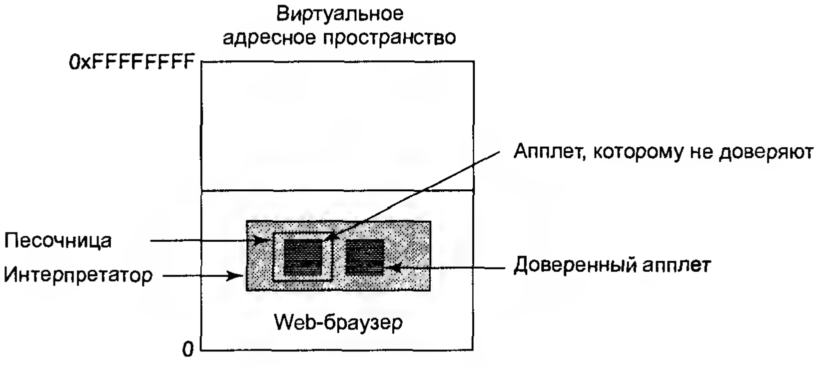


Рис. 9.12. Апплеты могут интерпретироваться web-браузером

Преимущество использования интерпретируемой программы перед исполняемой заключается в том, что каждая команда программы перед исполнением изучается интерпретатором. Это дает интерпретатору возможность проверить допустимость адресов. Кроме того, системные вызовы также перехватываются и интерпретируются. Обработка этих системных вызовов зависит от применяемой политики безопасности. Например, если апплету можно доверять (предположим, он поступил с локального диска), его системные вызовы могут выполняться без лишних вопросов. Однако если апплету доверять нельзя (скажем, если он пришел по Интернету), он может быть помещен в песочницу, чтобы ограничить его возможности.

Программы, написанные на высокоуровневых языках сценариев, также могут интерпретироваться. В этом случае машинные адреса не используются, поэтому нет опасности, что сценарий попытается получить доступ к памяти недопустимым образом. Недостаток интерпретации в основном состоит в том, что такой метод исполнения программ значительно медленнее по сравнению с исполнением откомпилированных программ.

[1] Точнее, JVM — это не язык, а нечто вроде операционной системы. Так называется сам интерпретатор, выполняющий апплеты Java. — *Примеч. перев.*

Программы с подписями

Еще один способ обеспечения безопасности исполнения апплетов заключается в том, что клиент знает, откуда пришел апплет, и запускает только апплеты от доверенных источников. При таком подходе пользователь может содержать список доверенных поставщиков апплетов и запускать апплеты только этих поставщиков. Апплеты ото всех остальных источников отвергаются как ненадежные. При таком подходе никаких механизмов обеспечения безопасности во время исполнения не используется. Апплеты от доверенных поставщиков запускаются в том виде, в котором они получены, а все остальные апплеты не запускаются вообще, либо запускаются в стреноженном виде (с помощью песочниц или интерпретации с ограниченным доступом или вообще без доступа к файлам пользователя и другим системным ресурсам).

Чтобы такая схема работала, у пользователя как минимум должен быть способ определить, что апплет был написан производителем, которому можно доверять, и не изменен кем-либо еще впоследствии. Это может реализовываться при помощи цифровых подписей.

Для цифровых подписей используется шифрование с открытым ключом. Производитель апплета, как правило компания, занимающаяся производством программного обеспечения, создает пару (открытый ключ, закрытый ключ), публикует открытый ключ и тщательно охраняет закрытый. Чтобы подписать апплет, производитель сначала вычисляет хэш-код апплета, 128-разрядный или 160-разрядный, в зависимости от используемого алгоритма (MD5 или SHA). Затем хэш-код зашифровывается (или «расшифровывается», если применять нотацию рис. 9.2). Эта подпись сопровождает апплет, куда бы тот ни направлялся.

Когда пользователь получает апплет, браузер вычисляет значение хэш-функции. Затем он расшифровывает открытым ключом прилагающуюся подпись и сравнивает два хэш-кода. Если они совпадают, апплет принимается. В противном случае он считается подделкой. Используемая математика делает крайне сложной подделку апплета или подписи, если не известен закрытый ключ. Процесс формирования и проверки подписи проиллюстрирован на рис. 9.13.

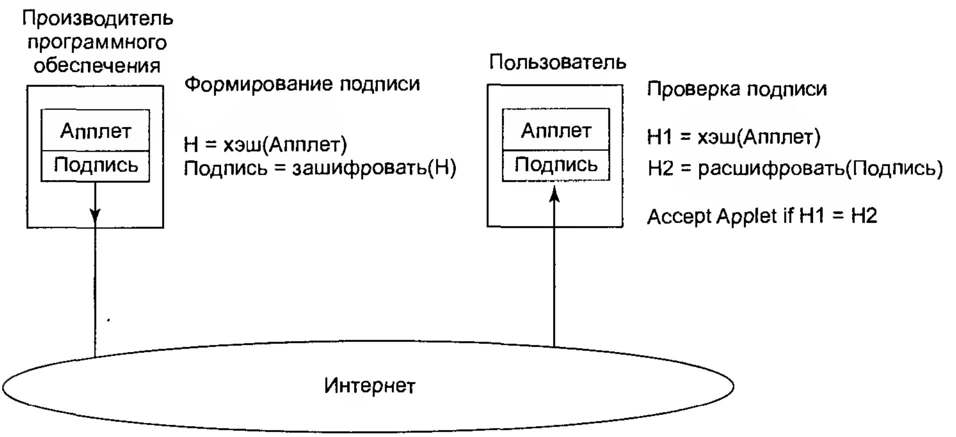


Рис. 9.13. Принцип работы цифровой подписи

Безопасность в системе Java

Язык программирования Java и соответствующая интерпретирующая система были разработаны, чтобы можно было один раз написать и откомпилировать программу, которая потом могла бы загрузиться по Интернету и выполниться на любой машине, поддерживающей Java. Безопасность являлась частью системы Java с самого начала. В данном разделе будут описаны принципы работы системы безопасности Java.

Java представляет собой язык программирования, обеспечивающий типовую безопасность. Это означает, что компилятор отвергнет любые попытки использовать переменную каким-либо способом, несовместимым с ее типом. Для сравнения рассмотрим пример программы на языке С:

```
naughty_func()
{
char *p;
p = rand();
*p = 0;
}
```

Эта функция формирует псевдослучайное число (целое) и сохраняет его в указателе на байтовый массив *p*[1]. Затем по новому адресу, содержащемуся в переменной *p*, перезаписывается все, что там было, программа или данные. В языке Java конструкции, смешивающие типы подобным образом, запрещены на уровне грамматики. Кроме того, в языке Java нет переменных указателей, преобразования типов, управляемых пользователем процедур выделения памяти (таких, как *malloc* и *free*), а обращения к массивам проверяются во время исполнения программы.

Программы на языке Java компилируются в двоичный код, называемый **JVM-кодом** (Java Virtual Machine — виртуальная машина Java). В этом коде около 100 команд, большая часть которых помещает объекты определенного типа в стек, достает их из стека или выполняет с объектами, хранящимися в стеке, арифметические действия. Как правило, эти программы интерпретируются, хотя в некоторых случаях они могут компилироваться в машинный код для более быстрого исполнения. В модели Java апплеты, посылаемые по Интернету для удаленного выполнения, являются JVM-программами.

Когда апплет прибывает, он пропускается сквозь процедуру, проверяющую, удовлетворяет ли апплет определенным правилам. Откомпилированный соответствующим образом апплет автоматически подчиняется этим правилам, ничто не мешает злоумышленнику написать JVM-апплет на JVM-ассемблере. В проверку входит:

1. Пытается ли апплет подделать указатели?
2. Нарушает ли он ограничения доступа к закрытым членам классов?
3. Пытается ли он использовать переменную одного типа как переменную другого типа?
4. Не переполняет ли апплет стек и не пытается ли он переместить указатель стека за его предельные границы?
5. Не преобразует ли он незаконно переменные из одного типа в другой?

---

[1] Такое было возможно, но только в очень древнем компиляторе С. В Visual C++ 5-й и 6-й версий компилятор откажется транслировать такую программу, требуя явного преобразования типов. — *Примеч. перев.*

Если апплет проходит все тесты, его можно смело запускать, не опасаясь, что он обратится к области памяти за пределами апплета.

Однако апплеты могут обращаться к системным вызовам, вызывая методы (процедуры) Java, предоставленные для этой цели. В первой версии системы Java, **JDK 1.0** (Java Development Kit — инструментальный комплект поддержки разработок в среде Java), апплеты подразделялись на два класса: доверенные и те, которым доверять нельзя. Апплеты, загружаемые с локального диска, считались доверенными, и им позволялось обращаться к любым системным вызовам. К апплетам, загружаемым из Интернета, напротив, относились с недоверием. Эти апплеты помещались в песочницы, как показано на рис. 9.12, и им практически ничего не разрешалось делать.

Поэкспериментировав некоторое время с этой моделью, корпорация Sun пришла к выводу, что она слишком сильно ограничивает возможности апплетов. В пакете JDK 1.1 были использованы электронные подписи. Когда апплет прибывал по Интернету, виртуальная машина Java проверяла, был ли он подписан человеком или организацией, которой пользователь доверял (что указывалось пользователем в списке доверенных поставщиков апплетов). Если у апплета была соответствующая подпись, ему разрешалось выполнение. В противном случае он помещался в песочницу со строгими ограничениями.

Получив больше опыта, корпорация Sun убедилась, что этих мер защиты также недостаточно, поэтому модель безопасности была снова изменена. Пакет JDK 1.2 предоставляет возможность детальной настройки политики безопасности, применимой ко всем апплетам, как к локальным, так и к удаленным. Модель безопасности, применяемая в данном пакете, настолько сложна, что ей может быть посвящена отдельная книга [131], поэтому здесь мы ограничимся кратким перечислением наиболее ярких моментов.

Каждый апплет характеризуется двумя параметрами: откуда он пришел и кто его подписал. Откуда он пришел, указано в его URL-указателе. Каждый пользователь может создать политику безопасности, состоящую из набора правил. В правиле могут перечисляться URL-указатель, владелец подписи, объект и действие, которые апплету разрешается выполнять с этим объектом, если URL и владелец подписи соответствуют указанным в правиле. Пример информации, содержащейся в правиле, продемонстрирован в табл. 9.3, хотя фактическая форма представления этих данных может быть другой и относится к иерархии классов Java.

Таблица 9.3. Несколько примеров политик безопасности пакета JDK 1.2

URL	Владелец подписи	Объект	Действие
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Чтение
*		/usr/tmp/*	Чтение, запись
www.microsoft.com	Microsoft	/usr/susan/Office/–	Чтение, запись, удаление

Действия с объектами могут включать доступ к файлам. Действие может указывать конкретный файл или каталог, все файлы в заданном каталоге или все файлы и каталоги, содержащиеся в данном каталоге. Этим трем случаям соответствуют три строки в табл. 9.3. В первой строке пользователь Сьюзан разрешила чтение

файла *1040.xls* для апплетов, загружающихся с машины www.taxprep.com, принадлежащей компании TaxPrep и подписанных этой компанией. Это единственный файл, который разрешается читать этому апплету, и никакой другой апплет не может читать этот файл. Кроме того, всем апплетам, подписанным или нет, разрешается читать и писать файлы в каталоге */usr/tmp*.

Сьюзан также настолько доверяет корпорации Microsoft, что позволяет всем апплетам с сайта этой компании читать, писать и удалять все файлы в каталоге *Office* со всеми его подкаталогами, например, чтобы исправлять ошибки или устанавливать новые версии программного обеспечения. Для проверки подписей Сьюзан нужно либо иметь все необходимые открытые ключи на своем диске, либо она должна получать их динамически, например, в виде сертификатов, подписанных компанией, которой она доверяет и чьи открытые ключи у нее уже есть.

Защищаться могут не только файлы. Доступ к сети также может быть защищен. Объектами в данном случае являются определенные порты или конкретные компьютеры. Компьютер указывается IP-адресом или DNS-именем. Порты на этой машине указываются в виде диапазона чисел. К возможным действиям относятся запрос соединения с удаленной машиной и прием соединений, исходящих от удаленного компьютера. Таким образом, апплету может быть предоставлен сетевой доступ, ограниченный списком указанных компьютеров и портов. Апплеты могут динамически подгружать дополнительные программы (классы), но предоставляемые пользователем загрузчики классов имею право указывать, с каких машин данные классы могут быть загружены. Имеется и множество других настраиваемых параметров безопасности.

Механизмы защиты

В предыдущих разделах мы рассмотрели множество проблем, некоторые из них были техническими, тогда как другие — нет. В следующих разделах мы сконцентрируемся на некоторых технических деталях методов защиты файлов, используемых в операционных системах. Во всех этих методах проводится четкое разграничение между политикой (от кого и чьи данные должны защищаться) и механизмом (как система проводит данную политику). Отделение политики от механизма обсуждается в [289]. Мы уделим особое внимание именно механизму, а не политике.

В некоторых системах защита реализуется при помощи программы, называющейся **монитором обращений**. При каждой попытке доступа к некоторому ресурсу система сначала просит монитор обращений проверить законность данного доступа. Монитор обращений смотрит в таблицы политики и принимает решение. Ниже будет описано окружение, в котором работает монитор обращений.

Домены защиты

Компьютерная система содержит множество «объектов», которые требуется защищать. Это может быть аппаратура (например, центральный процессор, сегменты памяти, диски или принтеры) или программное обеспечение (например, процессы, файлы, базы данных или семафоры).

У каждого объекта есть уникальное имя, по которому к нему можно обращаться, и набор операций, которые могут выполнять с объектом процессы. Так, с файлом могут выполняться операции read и write, с семафором имеют смысл операции up и down.

Очевидно, что для ограничения доступа к объектам требуется определенный механизм. Более того, этот механизм должен предоставлять возможность не полного запрета доступа, а ограничения в пределах подмножества разрешенных операций. Например, процессу *A* может быть разрешено читать, но не писать файл *F*.

Чтобы обсудить различные механизмы защиты, полезно ввести концепцию домена. **Домен** представляет собой множество пар (объект, права доступа). Каждая пара указывает объект и некоторое подмножество операций, которые могут быть с ним выполнены. **Права доступа** означают в данном контексте разрешение выполнить одну из операций. Домен может соответствовать одному пользователю или группе пользователей.

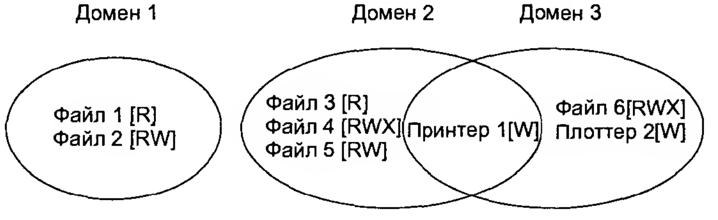


Рис. 9.14. Три домена защиты

На рис. 9.14 показаны три домена, содержащие объекты и разрешения [RWX — Read, Write, eXecute — чтение, запись выполнение] для каждого объекта. Обратите внимание, что объект *Printer1* одновременно присутствует в двух доменах. Хотя это и не изображено в данном примере, один и тот же объект может иметь в различных доменах разные разрешения.

В каждый момент времени каждый процесс работает в каком-либо одном домене защиты. Другими словами, имеется некоторая коллекция объектов, к которым он может получить доступ, и для каждого объекта у него есть определенный набор разрешений. Во время выполнения процессы имею право переключаться с одного домена на другой. Правила переключения между доменами в большой степени зависят от системы.

Чтобы идея домена защиты выглядела более конкретно, рассмотрим пример из системы UNIX. В UNIX домен процесса определяется его идентификаторами UID и GID процесса. По заданной комбинации (UID, GID) можно составить полный список всех объектов (файлов, включая устройства ввода-вывода, представленные в виде специальных файлов и т. д.), к которым процесс может получить доступ с указанием типа доступа (чтение, запись, исполнение). Два процесса с одной и той же комбинацией (UID, GID) будут иметь абсолютно одинаковый доступ к одинаковому набору объектов.

Более того, каждый процесс в UNIX состоит из двух частей: пользовательской части и системной части. Когда процесс обращается к системному вызову, он переключается из пользовательской части в системную. Пользовательская и системная части процесса имеют различный доступ к различным множествам объектов. Например, системная часть процесса может иметь доступ ко всем страницам

в физической памяти, ко всему диску и ко всем другим защищенным ресурсам. Таким образом, системный вызов осуществляет переключение доменов защиты.

Когда процесс выполняет системный вызов exec с файлом, у которого установлен бит SETUID или SETGID, процесс может получить новые идентификаторы UID и GID. При новой комбинации UID и GID процесс получает новый набор доступных файлов и операций. Запуск программы с установленным битом SETUID или SETGID также представляет собой переключение домена, так как права доступа при этом изменяются.

Важным вопросом является то, как система отслеживает, какой объект какому домену принадлежит. Можно себе представить большую **матрицу**, в которой рядами являются домены, а колонками — объекты. На пересечении располагаются ячейки, содержащие права доступа для данного домена к данному объекту. Матрица для табл. 9.3 показана на рис. 9.15. При наличии подобной матрицы операционная система может для каждого домена определить разрешения к любому заданному объекту.

Домен \ Объект	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2
1	Чтение	Чтение Запись						
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись	
3						Чтение Запись Исполнение	Запись	Запись

Рис. 9.15. Матрица защиты

Переключение между доменами также может быть легко реализовано при помощи все той же матрицы, если считать домены объектами, над которыми возможна операция enter (вход). На рис. 9.16 снова изображена та же матрица, что и на предыдущем рисунке, но с тремя доменами, выступающими и в роли объектов. Процессы могут переключаться с домена 1 на домен 2, но обратно вернуться уже не могут. Эта ситуация моделирует выполнение программы с установленным битом SETUID в UNIX. Другие переключения доменов в данном примере не разрешены.

Домен	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2	Домен 1	Домен 2	Домен 3
1	Чтение	Чтение Запись								Enter	
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Чтение				
3						Чтение Запись Исполнение	Чтение	Чтение			

Рис. 9.16. Матрица защиты с доменами в роли объектов

Списки управления доступом

На практике, однако, разрешения доступа редко хранятся в виде матрицы, показанной на рис. 9.16, поскольку такая матрица была бы очень большой и практически пустой. Поэтому, как правило, такие матрицы хранятся по рядам или столбцам, причем хранятся только непустые элементы. Эти два подхода, как ни странно, различаются между собой. В данном разделе мы рассмотрим хранение матрицы защиты по столбцам, в следующем разделе мы познакомимся с методом ее хранения по рядам.

В первом методе с каждым объектом ассоциируется список (упорядоченный), содержащий все домены, которым разрешен доступ к данному объекту, а также тип доступа. Такие списки, называемые **ACL**-списками (Access Control List — список управления доступом), проиллюстрированы на рис. 9.17. Здесь мы видим три процесса, принадлежащих различным доменам *A*, *B* и *C*, а также три файла, *F*1, *F*2 и *F*3. Для простоты мы будем предполагать, что каждому домену соответствует один пользователь, в данном случае *A*, *B* и *C*. Часто в литературе по информационной безопасности пользователей называют **субъектами** или **принципалами** (владельцами объектов), чтобы отличать их от **объектов**, которыми кто-то владеет.

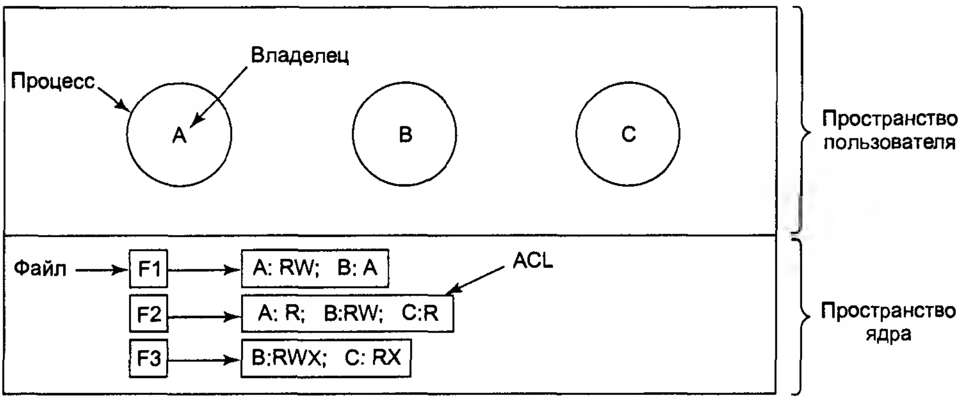


Рис. 9.17. Использование ACL-списков для управления доступом к файлам

С каждым файлом связан ACL-список. У файла *F*1 в его ACL-списке есть три записи (разделенные символом точка с запятой). В первой записи утверждается, что любой процесс, которым владеет пользователь *A*, может читать и писать этот файл. Во второй записи говорится, что любой процесс, которым владеет пользователь *B*, может читать этот файл. Все остальные типы доступа для данных пользователей запрещаются. Всем остальным пользователям запрещен любой доступ к этому файлу. Обратите внимание, что права предоставляются не процессу, а пользователю. Таким образом, любой процесс, которым владеет пользователь *A*, может читать и писать файл *F*1. Не имеет значения, один такой процесс или их 100. Значение имеет идентификатор владельца, а не процесса.

У файла *F*2 в его ACL-списке есть три записи: пользователи *A*, *B* и *C* могут читать файл, а пользователь *B* также может писать в него. Другие типы доступа не разрешаются. Файл *F*3, по-видимому, представляет собой исполняемую программу,

так как пользователи *A* и *B* могут как читать, так и исполнять его. Пользователю *B* также разрешается писать в этот файл.

Этот пример иллюстрирует самую общую форму защиты при помощи ACL-списков. Часто на практике используются более сложные системы. Начнем с того, что мы здесь показали только три типа доступа: чтение, запись и исполнение. Кроме них может быть еще много других типов доступа. Некоторые типы доступа могут быть применимы ко всем объектам, например уничтожение или копирование объекта, а некоторые могут быть специфическими для определенных объектов. Примеры подобных разрешений: `добавить сообщение` для объекта почтовый ящик и `сортировать в алфавитном порядке` для объекта каталог.

До сих пор мы рассматривали ACL-списки индивидуальных пользователей. Многими системами поддерживается концепция **групп** пользователей. У групп есть имена, и они также могут включаться в ACL-списки. Возможно применение двух вариантов семантики групп. В некоторых системах у каждого процесса есть идентификатор пользователя UID и идентификатор группы GID. В других системах ACL-списки содержат записи вида

```
UID1.GID1: права1; UID2.GID2: права2; ...
```

При такой схеме, когда к объекту поступает запрос доступа, выполняется проверка с помощью UID и GID обращающегося с запросом процесса. Если они присутствуют в ACL-списке, то перечисленные в списке права предоставляются процессу. Если комбинации (UID, GID) нет в списке, в доступе к объекту отказывается.

Использование групп вводит понятие **роли**. Рассмотрим систему, в которой Тана является системным администратором и поэтому входит в группу *sysadm*. Однако предположим, что в компании есть также клубы для сотрудников и Тана является членом клуба любителей голубей. Члены клуба принадлежат к группе *pigfan* и обладают доступом к компьютерам компании для управления своей голубиной базой данных. Часть ACL-списка может быть показана в табл. 9.4.

Таблица 9.4. Два элемента ACL-списка

Файл	Список управления доступом
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Если Тана пытается получить доступ к одному из этих файлов, результат зависит от того, под какой учетной записью она зарегистрировалась в системе. Во время регистрации система может попросить ее выбрать группу, которую она намеревается использовать. Также Тана может зарегистрироваться в системе под различными именами. Цель этой схемы состоит в том, чтобы Тана не могла иметь доступа к файлу паролей, когда занимается голубями. Доступ к файлу паролей она может получить только в том случае, когда регистрируется в системе как системный администратор.

В некоторых случаях пользователю может предоставляться доступ к определенным файлам независимо от того, к которой группе он принадлежит в данный

момент. Эта ситуация в записи ACL-списка может обозначаться, например, символом **звездочки**, означающей все группы. Так, запись

```
tana,* :RW
```

для файла паролей предоставила бы Тане доступ к нему независимо от того, к которой группе Тана принадлежит в данный момент.

Кроме того, права доступа могут предоставляться пользователю, принадлежащему к определенной группе, если эти права предоставлены всей группе. В данном случае пользователь, принадлежащий одновременно к нескольким группам, не должен задумываться, какую группу указывать во время регистрации. При такой схеме членство в группах является постоянным и не указывается при регистрации в системе. Недостаток такого подхода заключается в том, что он предоставляет меньшую степень инкапсуляции: Тана может редактировать файл паролей во время собрания голубиного клуба.

С помощью групп и символов звездочки можно селективно блокировать определенного пользователя, отказав ему в доступе к конкретному файлу. Например, запись

```
hacker,* : (none); *,* :RW
```

предоставляет разрешения чтения и записи файла всем пользователям, кроме пользователя hacker. Это работает, так как записи сканируются слева направо и выбирается первая подходящая; последующие записи даже не изучаются. Следовательно, находится запись hacker и соответствующий ей уровень доступа (none), то есть никакого. На этом поиск прекращается.

Другая схема работы с группами заключается в том, что записи ACL-списка состоят не из пар (UID, GID), а просто содержат только UID или GID. Например, запись для файла *pigeon_data* может выглядеть следующим образом:

```
debbie: RW; phil: RW; pigfan: RW
```

означая, что Дебби и Фил, а также все члены группы *pigfan* могут читать и писать этот файл.

Иногда бывает так, что у какого-либо пользователя или группы есть определенные разрешения доступа к файлу, которые владелец файла хотел бы у них отнять. С помощью списков управления доступом аннулировать предоставленные ранее права доступа довольно просто. Все, что для этого нужно, — это отредактировать ACL-список, чтобы внести соответствующие изменения. Однако если ACL-список проверяется только при открытии файла, вероятнее всего, эти изменения затронут только последующие системные вызовы open. Любой уже открытый файл будет сохранять права на этот файл, полученные при его открытии, даже если пользователю вообще запретили любой доступ к этому файлу, но уже после того, как он открыл файл.

Перечни возможностей

Матрица, показанная на рис. 9.16, может также храниться по рядам. При использовании этого метода с каждым процессом ассоциирован список объектов, к которым может быть получен доступ, вместе с информацией о том, какие операции

разрешены с каждым объектом, другими словами, доменом защиты объекта. Такой список называется **перечнем возможностей**, а его элементы называются **возможностями** [93, 113]. Пример трех процессов и их перечней возможностей показан на рис. 9.18.

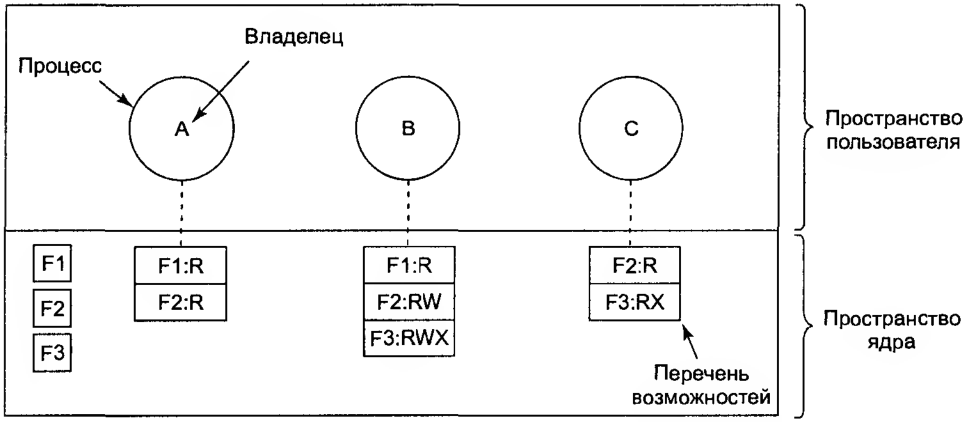


Рис. 9.18. Использование ACL-списков для управления доступом к файлам

Каждый элемент перечня возможностей предоставляет владельцу определенные права по отношению к определенному объекту. Например, на рис. 9.18 процесс, которым владеет пользователь *A*, может читать файлы *F*1 и *F*2. Обычно элемент перечня возможностей состоит из идентификатора файла (или, в более общем случае, объекта) и битового массива для различных разрешений. В операционных системах типа UNIX в качестве идентификатора файла может использоваться номер его i-узла. Перечни возможностей сами являются объектами и на них могут указывать другие перечни возможностей, таким образом облегчая совместное использование субдоменов.

Очевидно, что перечни возможностей должны быть защищены от искажения их пользователями. Известны три способа их защиты. Для первого способа требуется **теговая архитектура**, то есть аппаратно реализованная структура памяти, в которой у каждого слова памяти есть дополнительный (теговый) бит, сообщающий, содержит ли данное слово памяти элемент перечня возможностей или нет. Теговый бит не используется в обычных командах процессора, таких как арифметические или команды сравнения. Изменен он может быть только программой, работающей в режиме ядра (то есть операционной системой). Машины с подобной архитектурой были построены и хорошо себя показали [116]. В качестве популярного примера такой машины можно называть компьютер IBM AS/400.

Второй способ состоит в том, что перечень возможностей хранится внутри операционной системы. При этом к элементам перечня возможностей можно обращаться по их позиции в перечне. Процесс может запросить, например, прочитать 1 Кбайт из файла, на который указывает элемент перечня возможностей номер 2. Такая форма адресации напоминает использование дескрипторов файла в UNIX. Именно таким образом работала система Hydra [365].

Третий способ заключается в хранении перечня возможностей в пространстве пользователя, но в зашифрованном виде, так чтобы пользователь не смог изменить эту информацию. Этот метод хорошо подходит для распределенных систем и работает следующим образом. Когда клиентский процесс отправляет сообщение удаленному серверу, например файловому серверу, чтобы создать объект для него, сервер создает объект, а также формирует большое случайное число, используемое как контрольное поле. В таблице файлового сервера для объекта резервируется запись, в которой также хранится контрольное поле, адреса дисковых блоков и т. д. В терминах системы UNIX контрольное поле хранится на сервере в i-узле. Оно не посылается пользователю и вообще никогда не попадает в сеть. Затем сервер формирует и передает пользователю элемент перечня возможностей, формат которого показан на рис. 9.19.

Сервер	Объект	Права	f(Objects,Rights,Check)

Рис. 9.19. Элемент перечня возможностей, защищенный с помощью шифрования

Возвращаемый пользователю элемент перечня возможностей содержит идентификатор сервера, номер объекта (индекс в таблицах сервера, по сути, i-узел), а также права доступа, хранящиеся в виде битового массива. У только что созданного объекта все биты разрешений установлены в единицу. Последнее поле представляет собой результат необратимой функции *f* от конкатенации объекта, прав и контрольного поля. Такая функция уже обсуждалась ранее.

Когда пользователь желает получить доступ к объекту, он отправляет в запросе серверу элемент перечня возможностей. Сервер извлекает из него номер объекта и использует его в качестве индекса для поиска объекта в своих таблицах. Затем он вычисляет *f*(*Objects*, *Rights*, *Check*), причем первые два параметра для этой функции он берет из присланного ему пользователем элемента перечня возможностей, а третий параметр из своих таблиц. Если результат совпадает с четвертым полем элемента перечня возможностей, запрос удовлетворяется, в противном случае он отвергается. Если пользователь пытается получить доступ к чужому объекту, он не сможет подделать четвертое поле, так как не знает значения контрольного поля.

Пользователь может попросить сервер создать элемент перечня возможностей с меньшими правами, например позволяющий только читать объект. Сначала сервер проверяет действительность элемента перечня возможностей. Если подпись совпадает, он вычисляет *f*(*Objects*, *New_rights*, *Check*) для нового разрешения и создает новый элемент перечня возможностей, помещая это значение в четвертое поле. Обратите внимание, что при этом используется оригинальное значение контрольного поля *Check*, так как от него зависят другие элементы перечня возможностей.

Этот новый элемент перечня возможностей посылается обратно запрашивающему процессу. Теперь пользователь может передать его, скажем, отправив по электронной почте своему другу. Если друг попытается установить в единицу какой-либо бит разрешения, сервер тут же обнаружит это, так как значение функции *f* изменится. Поскольку другу неизвестно значение контрольного поля, он не сможет подделать элемент перечня возможностей так, чтобы тот соответствовал

фальшивым битам разрешений. Эта схема была разработана для распределенной операционной системы Amoeba и активно в ней применялась [324].

Помимо специфических разрешений, зависящих от конкретного объекта, например чтение и исполнение, элементы перечня возможностей (как системные, так и защищенные шифрованием) содержат, как правило, **общие права**, то есть разрешения выполнения действий, применимых ко всем объектам. Примерами таких действий являются:

1. Копирование элемента перечня возможностей: создание нового элемента перечня возможностей для того же объекта.
2. Копирование объекта: создание дубликата объекта с новым элементом перечня возможностей.
3. Удаление элемента перечня возможностей: удаление записи в перечне возможностей; объект при этом не затрагивается.
4. Удаление объекта: удаление объекта и элемента перечня возможностей.

Напоследок стоит отметить, что аннулирование доступа к объекту в системах перечней возможностей, реализованных на уровне ядра, довольно сложно. Системе трудно найти все элементы перечня возможностей для конкретного объекта, чтобы забрать их, так как они могут храниться в перечнях возможностей по всему диску. Один из методов заключается в том, что элемент перечня возможностей должен указывать не на сам объект, а на косвенный объект. Система может в любой момент разорвать связь между объектом и указывающим на него косвенным объектом, таким образом, аннулируя все возможности. (Когда элемент перечня возможностей позднее появляется в системе, пользователь обнаружит, что косвенный объект теперь указывает на нулевой объект.)

В системе Amoeba аннулирование разрешений выполняется легко. Все, что для этого требуется, — это изменить контрольное поле, хранимое с объектом. В результате одного удара все существующие элементы перечня возможностей становятся недействительными. Однако ни одна схема не обеспечивает выборочного аннулирования разрешений, то есть невозможно, например, аннулировать разрешения Джона, не затронув всех остальных. Этот недостаток присущ всем системам перечней возможностей.

Другая проблема состоит в том, чтобы гарантировать, что владелец действительного элемента перечня возможностей не раздаст его копию 1000 своих лучших друзей. Эта проблема решается в системах типа Hydra, в которых перечнями возможностей управляет ядро, но подобное решение не работает в распределенных системах, таких как Amoeba.

С другой стороны, перечни возможностей позволяют очень элегантно решить проблему помещения мобильной программы в песочницу. При запуске чужая программа получает список возможностей, содержащий только те возможности, которые владелец машины согласен ей предоставить, например возможность вывода на экран и чтения и записи файлов из одного временного каталога, специально созданного для этой программы. Если мобильная программа помещается в собственный процесс только с этими ограниченными возможностями, она не сможет получить доступ к другим системным ресурсам. Таким образом, программа оказывается в песочнице, для чего не требуется модификация кода или интерпрета-

ция программы. Запуск программы с минимальным набором прав доступа, называемый **принципом наименьшего уровня привилегий**, представляет собой мощное средство при создании защищенных систем.

Итак, кратко подведем итоги. ACL-списки и перечни возможностей обладают взаимодополняющими свойствами. Перечни возможностей очень эффективны, так как если процесс выдает запрос вида «Откройте файл, на который указывает элемент перечня возможностей 3», то не требуется никакой дополнительной проверки. При использовании ACL-списков для открытия файлов может потребоваться процесс поиска (возможно, долгий). Если группы не поддерживаются системой, тогда, чтобы предоставить доступ чтения файла всем пользователям системы, потребуется перечислить всех пользователей в ACL-списке. Кроме того, перечни возможностей позволяют легко инкапсулировать процесс, чего не могут ACL-списки. С другой стороны, ACL-списки обеспечивают выборочное аннулирование разрешений, тогда как при использовании перечней возможностей это нереально. Наконец, если объект удаляется, а элемент перечня возможностей нет или наоборот, возникают проблемы, которых лишены ACL-списки.

Надежные системы

Большая часть этой главы была посвящена тому факту, что практически все современные компьютерные системы удерживают информацию так же надежно, как решето воду. Ежегодный ущерб, причиняемый вирусами и тому подобными проблемами, превышает один миллиард долларов, затрачиваемых на попытки восстановить поврежденные данные, не говоря уже об упущенной выгоде. Соответственно, возникают два очевидных вопроса:

1. Возможно ли создание защищенной компьютерной системы?
2. И если да, то почему она еще не создана?

Ответ на первый вопрос в целом будет положительным. Как построить защищенную систему, известно уже в течение нескольких десятилетий. Например, у системы MULTICS, разработанной в 1960 году, безопасность была одной из главных целей, и эта цель была достигнута.

Почему не создаются защищенные системы — вопрос более сложный. Здесь можно указать две фундаментальные причины. Во-первых, сегодняшние системы не являются защищенными, но пользователи не желают от них отказываться. Если бы корпорация Microsoft, например, объявила, что кроме системы Windows у нее есть новый продукт, некая защищенная высоконадежная операционная система SecureOS с врожденным иммунитетом ко всем вирусам, но без поддержки Windows-приложений, то совсем не очевидно, что все индивидуальные пользователи и компании тут же отказались бы от Windows и купили бы новую систему.

Вторая причина не столь проста. Единственный способ создать защищенную систему состоит в том, чтобы сохранить систему простой. Изобилие функций является злейшим врагом безопасности. Разработчики систем полагают (правы они или ошибаются — другой вопрос), что пользователи хотят видеть в системе все больше и больше разнообразных функций. Большее количество функций означа-

ет более высокий уровень сложности, соответственно, больше строк исходного текста, больше ошибок и больше дыр в системе безопасности.

Приведем два простых примера. В первых системах электронной почты сообщения посылались в виде ASCII-текста. Они были совершенно безопасны. Ничто в получаемом сообщении формата ASCII не могло повредить компьютерную систему. Затем кому-то в голову пришла идея передавать электронной почтой и другие типы документов, например файлы редактора *Word*, которые могут содержать макросы. Открытие такого документа означает запуск чужой программы на вашем компьютере. Независимо от того, какие песочницы применяются, запуск чужой программы на вашем компьютере опаснее, чем просмотр ASCII-текста. Требовали ли пользователи возможности переключения электронной почты с пассивных документов на активные программы? Возможно, нет, но разработчики систем подумали, что это будет замечательной идеей, не заботясь о связанных с этим проблемах безопасности.

Второй пример представляет собой такой же переход с пассивных на активные элементы в web-страницах. Когда Паутина состояла из пассивных HTML-страниц, она не представляла собой основной проблемы безопасности (хотя намеренно некорректная HTML-страница могла вызвать переполнение буфера). Теперь, когда многие web-страницы содержат программы (апплеты), запускаемые пользователем, чтобы просмотреть содержание web-страницы, все новые дыры в системе безопасности обнаруживаются чуть ли не каждый день. Как только одну дыру удается залатать, ее место занимает другая. Когда Паутина была полностью статичной, требовали ли пользователи динамического содержания? Автор этого не припомнит, но ввод динамических элементов в Паутину привел к появлению массы проблем безопасности. Похоже, что люди, ответственные за критическое отношение к нововведениям и обязанные в таких случаях говорить «Нет», просто уснули за штурвалом.

Как ни странно, еще остались некоторые организации, считающие, что хорошая система безопасности важнее, чем различные модные навороты. К таким организациям относятся, например, военные. В следующих разделах мы рассмотрим несколько вопросов, касающихся защищенных систем, суть которых можно выразить в одной фразе. Для построения защищенной системы следует использовать в ядре операционной системы модель безопасности, достаточно простую, чтобы разработчики были способны ее понять, а также следует сопротивляться давлению отойти от используемой модели под предлогом добавления в систему новых функций.

Высоконадежная вычислительная база

Люди, занимающиеся компьютерной безопасностью, чаще употребляют термин **надежная система** (trusted system), чем «защищенная система» (secure system). Надежная система — это система, в которой формально установлены требования безопасности и которая удовлетворяет этим требованиям. В сердце каждой надежной системы находится минимальная база **TCB** (Trusted Computing Base — высоконадежная вычислительная база), состоящая из аппаратного программного обеспечения, необходимого для проведения в жизнь всех правил безопасности. Если высоконадежная вычислительная база работает в соответствии с техническими условиями, безопасность системы не может быть нарушена независимо от любых других обстоятельств.

Высоконадежная вычислительная база TCB, как правило, состоит из большей части аппаратного обеспечения (кроме устройств ввода-вывода, не влияющих на безопасность), части ядра операционной системы и большей части или всех пользовательских программ, обладающих полномочиями суперпользователя (например, программы с установленным битом SETUID, владельцем которых является root в системе UNIX). К функциям операционной системы, которые должны быть частью TCB, относятся создание процесса, переключение процессов, управление картой памяти, а также часть файлового управления и управления вводом-выводом. В надежных системах база TCB часто представляет собой совершенно отдельную ото всей остальной операционной системы часть, что позволяет уменьшить ее размеры и проверить корректность работы.

Важную часть высоконадежной вычислительной базы составляет монитор обращений, как показано на рис. 9.20. Монитор обращений принимает все системные вызовы, имеющие отношение к безопасности, такие как открытие файлов, и принимает решение, следует их выполнять или нет. Таким образом, монитор обращений позволяет все решения о безопасности поместить в одном месте, не предоставляя возможности обойти их. Организация большинства операционных систем отличается от данной схемы, в чем заключается одна из причин их ненадежности.

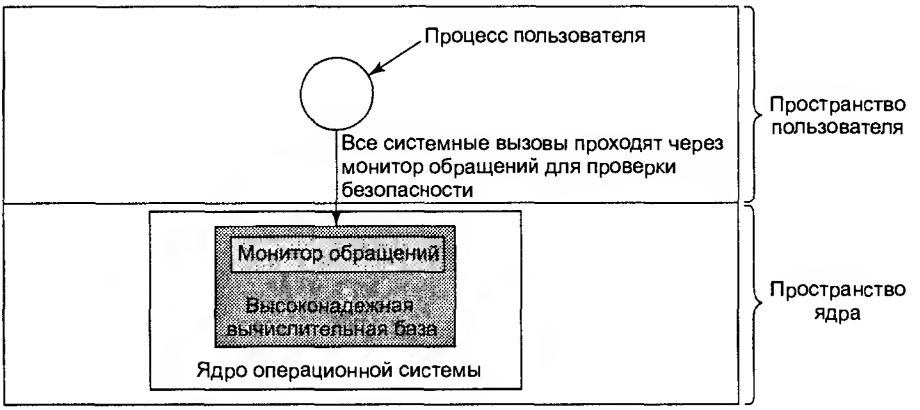


Рис. 9.20. Монитор обращений

Формальные модели защищенных систем

Матрицы защиты (см. рис. 9.15) нестатичны. Они часто меняются с созданием новых объектов, удалением старых объектов, а также когда владельцы объектов решают расширить или ограничить набор разрешений или круг пользователей, которым предоставляется доступ к объекту. Значительное внимание было уделено моделированию систем защиты, в которых матрицы защиты постоянно меняются. В оставшейся части данного раздела мы кратко рассмотрим некоторые из этих исследовательских работ.

Несколько десятилетий назад Харрисон и его коллеги [145] определили шесть примитивных операций над матрицей защиты, которые могут использоваться как основа для моделирования любой системы защиты. Эти примитивные операции представляют собой `создать объект`, `удалить объект`, `создать домен`, `удалить домен`, `доба-`

вить разрешение и удалить разрешение. Два последних примитива добавляют и удаляют разрешения в определенном элементе матрицы, например предоставляют разрешение читать файл *File6* домену 1.

Эти шесть примитивов можно объединять в **команды защиты**, которые могут выполняться программами пользователя для изменения матрицы. Пользовательские программы не могут напрямую выполнять примитивы. Например, в системе может быть команда для создания нового файла, которая проверяет, существует ли уже этот файл, и если нет, создает новый объект и предоставляет владельцу все права к нему. Также может быть команда, позволяющая владельцу файла передать права чтения этого файла любому другому пользователю, для чего в каждый домен вставляется запись, дающая право чтения этого файла.

В каждый момент времени матрица определяет, что может делать процесс в каждом домене, а не то, что ему разрешено делать. Матрица навязывается системой, разрешения относятся к политике управления. Чтобы это различие стало отчетливее, рассмотрим простой пример, в котором домены соответствуют пользователям (рис. 9.21). На рис. 9.21, *а* показана намеченная политика защиты: Генри может читать и писать данные в почтовом ящике *mailbox7*, Роберт может читать и писать файл *secret*, и все три пользователя могут читать и исполнять файл *compiler*.

	Объекты		
	Compiler	Mailbox 7	Secret
Эрик	Чтение Выполнение		
Генри	Чтение Выполнение	Чтение Запись	
Роберт	Чтение Выполнение		Чтение Запись

а

	Объекты		
	Compiler	Mailbox 7	Secret
Эрик	Чтение Выполнение		
Генри	Чтение Выполнение	Чтение Запись	
Роберт	Чтение Выполнение	Чтение	Чтение Запись

б

Рис. 9.21. Дозволенное состояние (*а*); недозволенное состояние (*б*)

Теперь представим себе, что Роберт очень умен и нашел способ подавать команды, изменяющие матрицу из состояния на рис. 9.21, *а* в состояние на рис. 9.21, *б*. При этом он получает доступ к файлу *mailbox7*. Если он попытается его прочитать, операционная система выполнит его запрос, ведь ей не известно, что состояние на рис. 9.21, *б* не является дозволенным.

Теперь должно быть ясно, что множество всех возможных матриц может быть разделено на два раздельных подмножества: множество всех дозволенных состояний и множество всех недозволенных состояний. Вопрос, которому было посвящено множество исследований, звучит следующим образом: «При заданном начальном дозволенном состоянии и наборе команд можно ли гарантировать, что система никогда не перейдет в недозволенное состояние?»

Таким образом, требуется ответить на вопрос о том, способен ли имеющийся механизм (набор команд защиты) провести в жизнь определенную политику защиты. Итак, у нас есть политика, начальное состояние матрицы и набор команд для модифицирования матрицы; при этом нам требуется способ гарантировать защищенность системы. Оказывается, такую защиту довольно трудно обеспечить. Многие операционные системы не являются защищенными даже в теории. Харри-

сон и его коллеги доказали, что в случае произвольной конфигурации для произвольной системы защиты защита является теоретически недостижимой [145]. Однако для конкретной системы можно доказать, что система может быть переведена из дозволенного состояния в недозволенное. Дополнительные сведения можно получить в [196].

Многоуровневая защита

Большинство операционных систем позволяют индивидуальным пользователям определять, кто может читать и писать их файлы, а также иметь доступ к другим их объектам. Такая политика называется **дискреционным управлением доступом**. Во многих окружениях эта модель прекрасно работает, но существуют среды, в которых необходимы значительно более жесткие требования к безопасности, как, например, в воинских частях, патентных отделах корпораций и больницах. В этих системах организации устанавливают правила, определяющие, кто и что может видеть, и эти правила не могут изменять солдаты, юристы или врачи, по крайней мере, без специального разрешения от своего начальства. Для таких систем, помимо стандартного дискреционного управления доступом, требуется **принудительное управление доступом**, чтобы гарантировать, что установленная политика безопасности реализуется системой. Принудительное управление доступом регулирует поток информации, гарантируя, что эта информация не потечет по маршруту, по которому ей течь не полагается.

Модель Белла–Ла Падулы

Из многоуровневых систем защиты самое широкое распространение получила **модель Белла–Ла Падулы**, поэтому мы начнем наше обсуждение многоуровневых систем защиты с нее [24]. Эта модель была разработана для обеспечения военной системы безопасности, но она также применима и в мирных целях. В военных организациях все документы (объекты) могут иметь уровни секретности, такие как несекретный, конфиденциальный, секретный и совершенно секретный. Военнослужащим также присваиваются различные уровни доступа к данной информации. Генералу может быть разрешено просматривать все документы, тогда как лейтенант может иметь доступ только к двум нижним уровням. Процесс, работающий в системе, приобретает уровень доступа запустившего его пользователя. Так как уровней секретности много, эта схема называется **многоуровневой системой безопасности**.

В модели Белла–Ла Падулы поток информации описывается следующими правилами:

1. **Простое свойство секретности.** Процесс, работающий на любом уровне секретности, может читать только объекты своего уровня или более низких уровней секретности. Например, генерал может читать документы лейтенанта, но лейтенант не может читать документы генерала.
2. **Свойство *.** Процесс, работающий на любом уровне секретности, может писать только в объекты своего уровня или более высоких уровней секретности. Например, лейтенант может добавить сообщение к почтовому ящику генерала, сообщая все, что ему известно, но генерал не может сделать то же самое с почтовым ящиком лейтенанта, так как ему могут быть известны сверхсекретные документы, которые не полагается знать лейтенанту.

Итак, в первом приближении процессы могут читать снизу и писать наверх, но не наоборот. Если система гарантированно реализует эти два свойства, можно доказать, что не будет утечки информации с уровня большей секретности на уровень меньшей секретности. Это свойство было названо *, потому что в своем оригинальном докладе авторы не смогли придумать для него подходящего названия и отложили задачу выдумывания имени на потом, решив временно использовать символ *. Они так и не придумали этого названия, и доклад был напечатан со звездочкой вместо него. В данной модели процессы читают и пишут объекты, но не общаются друг с другом напрямую. Графически модель Белла–Ла Падулы проиллюстрирована на рис. 9.22.

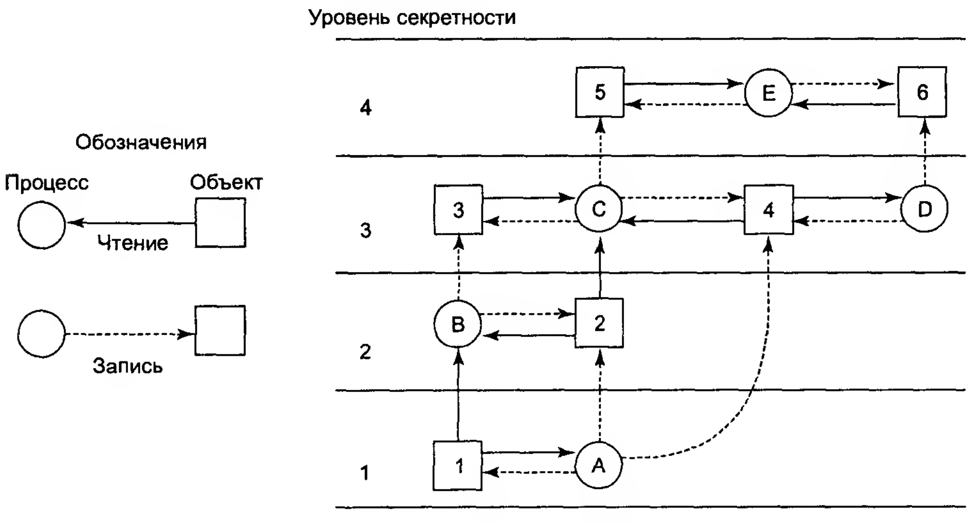


Рис. 9.22. Многоуровневая модель безопасности Белла–Ла Падулы

На данном рисунке сплошными стрелками от объекта к процессу показано направление информации при ее чтении процессом из объекта. Аналогично, штриховая стрелка от процесса к объекту означает запись данных в объект. Таким образом, вся информация течет по направлению стрелок. Например, процесс *B* может читать данные из объекта *1*, но не из объекта *3*.

Простое свойство секретности утверждает, что все сплошные стрелки (чтение) могут направляться в стороны или вверх. Свойство * говорит, что все штриховые стрелки (запись) могут также направляться в стороны или вверх. Поскольку информация распространяется только горизонтально или вверх, она не может попасть с более высокого уровня на более низкий. Другими словами, не может существовать пути информации сверху вниз, тем самым гарантируется защищенность модели.

Модель Биба

Итак, если перевести модель Белла–Ла Падулы на язык военных, лейтенант может приказать рядовому рассказать все, что ему известно, а затем скопировать эту информацию в файл генерала, не нарушив секретности. Теперь переведем эту модель

в термины гражданских лиц. Представим себе компанию, в которой у уборщиц уровень секретности 1, у программистов — уровень 3, а у президента компании уровень секретности 5. Используя модель Белла–Ла Падулы, программист может выведать у уборщицы стратегические планы компании, а затем перезаписать файл президента, содержащий стратегию корпорации. Не всем компаниям понравилась бы перспектива использования такой модели.

Недостаток модели Белла–Ла Падулы состоит в том, что она была разработана для хранения секретов, а не для гарантирования целостности данных. Чтобы гарантировать целостность данных, нам потребуются как раз противоположные свойства [32]:

1. **Простой принцип целостности.** Процесс, работающий на любом уровне секретности, может писать только в объекты своего уровня или более низких уровней секретности (запись в верхние уровни запрещена).
2. **Свойство целостности *.** Процесс, работающий на любом уровне секретности, может читать только в объекты своего уровня или более высоких уровней секретности (запрещено чтение низких уровней).

Вместе эти свойства гарантируют, что программист может изменять файлы уборщицы, записывая туда информацию, полученную от президента фирмы, но не наоборот. Конечно, некоторые организации хотели бы использовать одновременно и свойства модели Белла–Ла Падулы, и свойства модели Биба, но поскольку они противоречат друг другу, достичь этого сложно.

Оранжевая книга безопасности

Министерство обороны США также приложило определенные усилия к области развития систем безопасности. В частности, в 1985 году оно опубликовало документ, формально зарегистрированный как стандарт Министерства обороны США DoD 5200.28, но обычно называемый благодаря своей обложке **Оранжевой книгой**, в которой операционные системы подразделяются на семь категорий в зависимости от их свойств безопасности. Хотя с тех пор этот стандарт был заменен другим (намного более сложным), Оранжевая книга все еще представляет собой хорошее руководство в области безопасности систем. Кроме того, периодически можно встретить заявления производителей программного обеспечения о соответствии тому или иному уровню безопасности Оранжевой книги. Требования Оранжевой книги приведены в табл. 9.5. Ниже мы рассмотрим категории безопасности и выделим некоторые важные места. Символ X означает, что появились новые требования. Символ → означает, что на данном уровне применимы требования более низкой категории.

Уровни B и A требуют, чтобы всем управляемым пользователям и объектам были присвоены метки безопасности, такие как несекретный, секретный или сверхсекретный. Эта система должна поддерживать модель потоков информации Белла–Ла Падулы.

Уровень B2 добавляет к этому требование, заключающееся в том, что система должна разрабатываться сверху вниз в модульном виде. Конструкция системы

должна быть представлена в таком виде, чтобы ее можно было проверить. Должна быть проверена возможность наличия тайных каналов (см. следующий раздел).

Таблица 9.5. Критерии безопасности Оранжевой книги

Критерий	D	C1	C2	B1	B2	B3	A1
Политика секретности							
Дискреционное управление доступом		X	X	→	→	X	→
Повторное использование объекта			X	→	→	→	→
Метки				X	X	→	→
Целостность меток				X		→	→
Экспорт меченой информации				X		→	→
Маркировка вывода, удобочитаемого для человека				X		→	→
Принудительное управление доступом				X	X	→	→
Метки, чувствительные к предмету					X	→	→
Метки устройств					X	→	→
Возможность идентификации							
Идентификация и аутентификация		X	X	X			→
Аудит			X	X	X	X	→
Надежный путь					X	X	→
Гарантирование							
Архитектура системы		X	X	X	X	X	→
Целостность системы		X	→	→	→	→	→
Тестирование безопасности		X	X	X	X	X	X
Спецификация и верификация дизайна				X	X	X	X
Анализ тайных каналов					X	X	X
Надежное управление средствами					X	X	→
Управление конфигурацией					X	→	X
Надежное восстановление						X	→
Надежное распространение							X
Документация							
Руководство пользователя по безопасности		X	→	→	→	→	→
Руководство по надежным средствам		X	X	X	X	X	→
Тестовая документация		X	→	→	X	→	X
Документация разработчика		X	→	X	X	X	X

Уровень B3 содержит все свойства уровня B2, плюс он должен содержать ACL-списки с пользователями и группами; на нем также должны присутствовать формальная высоконадежная вычислительная база TCB, адекватный аудит безопасности и надежное восстановление от сбоев.

Уровень A1 требует наличия формальной модели системы защиты и гарантии корректности этой модели. Он также требует доказательств, что реализация системы соответствует модели. Тайные каналы должны быть формально проанализированы.

Тайные каналы

Все эти идеи о формальных моделях и гарантированно безопасных системах звучат замечательно, но работают ли они в действительности? Если ответить на этот вопрос одним словом, то нет. Даже в корректно реализованной системе, в основе которой лежит правильная модель безопасности и безопасность которой была доказана, могут случаться проколы в системе безопасности. В данном разделе мы обсудим, как информация может утекать даже в системах, для которых было строго математически доказано, что подобные утечки невозможны. Идеи, изложенные в данном разделе, были высказаны Лэмпсоном в [193].

Модель Лэмпсона изначально была сформулирована в терминах единой системы разделения времени, но те же идеи могут быть применены к локальной сети, а также другим многопользовательским средам. В ее чистом виде в эту модель входят три процесса, работающих на одной защищенной машине. Первый процесс представляет собой клиент, который хочет выполнить некоторое задание на втором процессе, сервере. Клиент и сервер доверяют друг другу не полностью. Например, работа сервера заключается в том, чтобы помочь клиентам заполнить их налоговые декларации. Клиенты беспокоятся, что сервер тайно запишет их финансовую информацию, а затем, например, продаст эти сведения. Сервер беспокоится, что клиенты украдут ценную программу подсчета налогов.

Третий процесс, называемый в данной модели «сообщником» (буквально collaborator, то есть «сотрудник»), намеревается украсть конфиденциальные сведения клиента. Владельцем этого процесса и сервера, как правило, является один и тот же человек. Все три процесса показаны на рис. 9.23. Цель данной модели состоит в том, чтобы разработать систему, в которой невозможна утечка информации, полученной сервером у клиента, к «сотрудничающему» процессу. Лэмпсон назвал это **проблемой ограждения**.

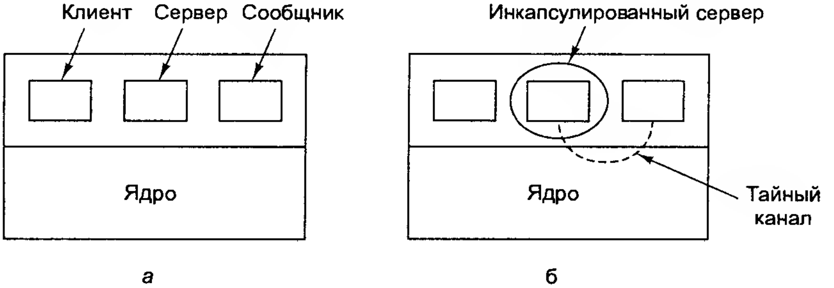


Рис. 9.23. Клиент, сервер и «сообщник» (*а*); от инкапсулированного сервера информация все равно может утекать «сообщнику» по тайному каналу (*б*)

С точки зрения разработчика системы задача заключается в инкапсуляции или ограждении сервера таким образом, чтобы он не мог передать информацию «сообщнику». С помощью схемы матрицы защиты несложно гарантировать, что сервер не сможет общаться с «сообщником», записывая данные в файл, к которому у «сообщника» есть доступ для чтения. Вероятно, можно также гарантировать,

что сервер не сможет общаться с «сообщником» при помощи системного механизма межпроцессного взаимодействия.

К сожалению, для утечки информации могут использоваться тайные каналы. Например, сервер может передавать последовательность нулей и единиц, кодируя единицы интервалами высокой активности процессора, а нули обозначая интервалами бездействия. «Сообщник» может попытаться принять этот поток, тщательно отслеживая время отклика сервера. Как правило, время отклика будет меньшим у простаивающего сервера, что соответствует нулю. Этот канал связи называется тайным каналом. Он показан на рис. 9.23, *б*.

Конечно, тайный канал представляет собой канал с шумом, но если использовать помехоустойчивое кодирование (например, код Хэмминга), информация может приниматься «сообщником» с высокой степенью надежности. Пропускная способность такого канала будет невелика, но это не снижает его опасности. Очевидно, что ни одна из моделей защиты, основанных на матрицах объектов и доменов, не сможет предотвратить данный тип утечки информации.

Модуляция использования центрального процессора не является единственным вариантом тайного канала. Информация также может кодироваться страничными прерываниями, например, много прерываний в течение определенного интервала времени будет означать 1, а отсутствие страничных прерываний — 0. В принципе для этой цели может использоваться любой способ снижения производительности системы в течение определенного интервала времени. Если система позволяет блокировать файлы, тогда сервер может блокировать некий файл, что будет означать, например, 1, и разблокировать его, кодируя, таким образом, 0. Некоторые системы позволяют процессу определить, что файл блокирован, даже если у него нет доступа к этому файлу. Такой тайный канал показан на рис. 9.24. В данном примере сервер передает сообщнику тайный поток бит 11010100.

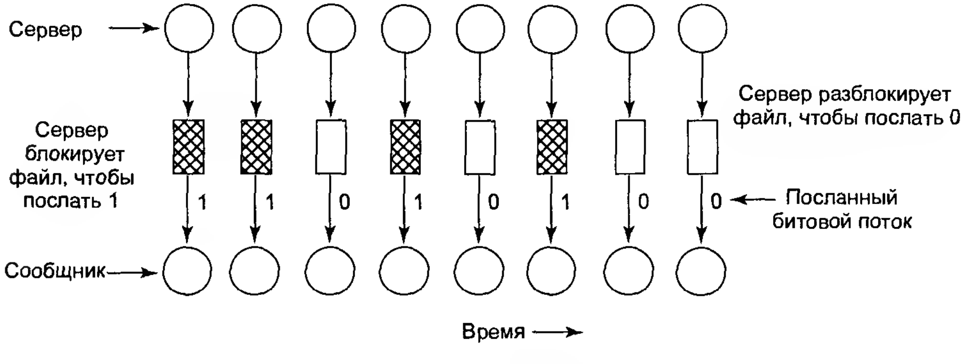


Рис. 9.24. Использование блокировки файла для тайного канала

Блокировка и разблокировка файла, о которых сервер и сообщник договорились заранее, не является особенно шумным каналом, однако для такого типа связи требуется особенно точная синхронизация, если только скорость передачи не очень низкая. Если использовать протокол с подтверждениями, то надежность и производительность такого тайного канала может быть значительно увеличена.

Для синхронизации сервер и сообщник открывают сразу три файла, *S*, *F*1 и *F*2, два из них блокируются сервером, а третий — сообщником. После того как сервер блокирует или разблокирует файл *S*, он изменяет состояние блокировки файла *F*1, сообщая тем самым сообщнику, что бит отправлен. Прочитав бит, сообщник изменяет в ответ состояние блокировки файла *F*2, подтверждая, что бит прочитан. Поскольку в данной схеме параметры времени не участвуют, этот протокол оказывается абсолютно надежным даже в системе, в которой одновременно запущено много процессов и сервер и центральный процессор сильно заняты. Скорость такого тайного канала данных зависит от того, насколько часто система переключает процессы. Повысить пропускную способность такого канала можно, если использовать одновременно два файла, *S*0 и *S*1, передавая сразу по два бита одновременно, или даже восемь файлов, передавая сразу один байт.

Для передачи скрытых сигналов также может использоваться захват и освобождение внешних ресурсов (например, магнитофонов, плоттеров и т. д.). В системе UNIX сервер может создавать файл, что будет означать 1, и удалять его, передавая 0. Сообщник может проверять наличие файла при помощи системного вызова access. Этот системный вызов будет работать, даже если у сообщника нет доступа к создаваемому сервером файлу. К сожалению, существует еще множество возможностей создания скрытого канала.

Лэмпсон также упомянул о еще одном возможном тайном канале связи, но уже между сервером и человеком. Например, сервер может сообщать, сколько работы он сделал для клиента, выставляя ему счет. Если настоящий счет составляет $100, а доход клиента составил $53 000, то сервер может сообщить об этом в виде счета в $100,53.

Обнаружить все скрытые каналы чрезвычайно трудно, не говоря уже об их блокировке. На практике можно сделать не слишком многое. Добавление процесса, вызывающего страничные прерывания случайным образом или снижающего производительность системы другим способом, чтобы снизить пропускную способность скрытых каналов, в качестве варианта решения проблемы не привлекает.

До сих пор мы предполагали, что клиент и сервер являются раздельными процессами. Другой вариант представляет собой только один работающий клиентский процесс, но при этом работающая клиентская программа является троянским конем. Такая схема может, например, применяться в том случае, когда система не позволяет сообщнику получить данные от сервера напрямую.

Существует еще одна интересная разновидность скрытого канала. Представьте себе компанию, которая вручную проверяет всю исходящую электронную почту, посылаемую сотрудниками, чтобы гарантировать, что никто не делится с конкурентами секретами фирмы. Есть ли способ, с помощью которого сотрудник компании может передать значительный объем конфиденциальной информации за пределы фирмы, прямо под носом цензора? Оказывается, есть.

Пример такого метода показан на рис. 9.25, *а*. На этой фотографии, снятой автором в Кении, мы видим трех зебр возле акации. Теперь взгляните на рис. 9.25, *б*. На первый взгляд, те же зебры и та же акация. Однако данная фотография, в отличие от первой, содержит также полный текст пяти пьес Шекспира, а именно:

«Гамлет», «Король Лир», «Макбет», «Венецианский купец» и «Юлий Цезарь». Все вместе они составляют более 700 Кбайт текста.

Как работает этот тайный канал? Оригинальное изображение состоит из 1024×768 пикселов. Каждый пиксел состоит из трех 8-разрядных чисел, по одному для интенсивности красного, зеленого и синего цветов пиксела. Цвет пиксела представляет собой линейную суперпозицию трех цветов. Для кодирования скрытого канала используются младшие разряды каждого значения цвета в формате RGB. Таким образом, в каждом пикселе помещается три бита секретной информации. В изображении данного размера может поместиться 1024×768×3 бит или 294 912 байт секретной информации.





а *б*

Рис. 9.25. Три зебры и дерево (*а*); три зебры, дерево и полный текст пяти пьес Шекспира (*б*)

Полный текст пяти пьес и короткого сообщения составляют 734 891 байт. При помощи стандартного алгоритма сжатия этот размер может быть уменьшен до 274 байт. Затем результат сжатия шифруется и помещается в младшие разряды каждого байта цвета. Как видно (то есть не видно), присутствие этой информации в изображении абсолютно незаметно на глаз. На большом цветном изображении этой информации также не видно. Человеческий глаз не в состоянии отличить 7-разрядный цвет от 8-разрядного. После того как эта фотография благополучно минует цензора, получатель просто считывает из файла младшие разряды, применяет алгоритмы дешифрации и распаковки и получает исходные 734 891 байт текста. Данный способ сокрытия информации называется **стеганографией** (что по-гречески означает «скрытая надпись»). Стеганографию не любят правительства, пытающиеся ограничить общение между их гражданами, но этот метод весьма популярен среди людей, верящих в свободу слова.

Конечно, два черно-белых изображения низкого разрешения не могут дать полного представления о мощи данного метода. Чтобы предоставить читателю более убедительное доказательство, автор подготовил набор файлов, включая изображение, показанное на рис. 9.25, *б*, которые можно скачать с web-сайта www.cs.vu.nl/~ast/. Щелкните мышью на ссылке covered writing под заголовком STEGANOGRAPHY DEMO. Затем выполните указанные на этой странице инструкции для загрузки изображения и программ, необходимых для извлечения пьес.

Кроме того, стеганография может применяться для добавления к изображениям скрытых «водяных знаков». Подобный метод позволяет обнаружить и доказать факт кражи изображений с web-страницы и их повторного использования. Если окажется, что ваше изображение содержит скрытое сообщение, например, «Copyright 2000, General Images Corporation», то вам, скорее всего, будет очень трудно убедить судью, что это изображение вы изготовили сами. Данный метод также может использоваться для защиты музыки, фильмов и т. д.

Конечно, подобные «водяные знаки» можно попытаться удалить. Например, водяные знаки на изображении полностью уничтожаются, если повернуть изображение по часовой стрелке на один градус, затем применить преобразование с потерей информации, скажем, JPEG, а затем повернуть изображение снова на один градус, но влево. Наконец, изображение можно снова конвертировать в его исходный формат (например, GIF, BMP или TIF). Преобразование с потерей данных JPEG перемешает все младшие разряды, а поворот изображения добавит ошибок округления при операциях с плавающей точкой, что также добавит шума к младшим разрядам. Люди, пытающиеся защитить свои цифровые данные водяными знаками, знают это, и поэтому они пытаются применять схемы с избыточностью, а также применяют другие схемы, кроме использования младшего бита. В ответ на это взломщики также пытаются применять более изощренную технику и т. д.

Исследования в области безопасности

Компьютерная безопасность представляет собой тему, которой посвящено очень много исследований, однако большая их часть не связана напрямую с операционной системой. Основные исследования в этой области затрагивают такие темы, как безопасность сетей (например, электронной почты, Паутины, а также безопасность электронной коммерции), криптографию, Java или просто надежное управление компьютерной системой.

Однако также проводились исследования, более близкие к теме нашей книги. Например, все еще сохраняет актуальность проблема аутентификации пользователей. Монроуз и Рубин [238] исследовали проблему распознавания пользователей по динамике нажатий на клавиши, Пентланд и Чаудхури [262] приводят аргументы в пользу распознавания лиц пользователей, а Марк [223] среди прочих разработал метод моделирования данной схемы.

Вот еще несколько работ, относящихся к безопасности операционных систем. Бершад с соавторами [27] отстаивают свою точку зрения, заключающуюся в том, что защита является вопросом программного обеспечения, а не аппаратного обеспечения (то есть MMU). Мазирес и его коллеги [228] рассматривают защищенные распределенные файловые системы. Майерс и Лисков [242] изучают модели надежных информационных потоков. Чейз с соавторами [60] исследуют безопасность в системах с большим адресным пространством, занятым большим количеством процессов. Безопасности смарт-карт посвящена работа Кларка и Хоффмана [69]. Гольдберг и другие исследователи [129] занимались конструированием филогенезов вирусов.

Резюме

Операционные системы могут подвергаться различным атакам, от атак изнутри до вирусных атак. Многие атаки начинаются с того, что взломщик пытается просто угадать пароль. Для таких атак часто используются словари наиболее употребляемых паролей. Подобные атаки часто бывают удивительно успешными. Безопасность паролей может быть усилена при помощи так называемой «соли», одноразовых паролей, а также схемы «оклик-отзыв». Также могут применяться смарт-карты и биометрические индикаторы. На практике часто используется сканирование сетчатки глаза.

Существует много разновидностей атак операционной системы, включая троянского коня, фальшивые программы регистрации, логические бомбы, потайные входы и атаки, использующие переполнение буфера. Кроме того, могут использоваться такие методы, как запрашивание страниц памяти и считывание информации, оставшейся в них, обращение к запрещенным системным вызовам и даже попытки обмана или подкупа сотрудников с целью выведать секретную информацию.

Все большую проблему последнее время представляют собой вирусы. Существует большое разнообразие форм вирусов, включая вирусы, резидентные в памяти, вирусы, заражающие загрузочный сектор диска, а также макровирусы. Использование антивирусной программы, ищущей зараженные файлы по определенным последовательностям байт, полезно, но серьезные вирусы могут зашифровать большую часть своего кода, а также модифицировать остальную часть при реплицировании, что очень сильно усложняет их обнаружение. Некоторое антивирусное программное обеспечение не ищет определенные куски вирусов, а пытается поймать их за подозрительными действиями. Лучшим средством против вирусов является предохранение от вирусов. Поэтому не загружайте и не запускайте программ, авторство которых неизвестно и доверие к которым под вопросом.

В последние годы все большую популярность приобретают мобильные программы, например активные web-страницы. К возможным средствам борьбы с потенциальной опасностью таких программ относятся помещение мобильной программы в «песочницу», интерпретация программы, а также запуск только программ, подписанных доверенными производителями.

Операционные системы могут защищаться с помощью матриц, состоящих из доменов защиты (например, пользователей) по вертикали, объектов по горизонтали. Матрица может храниться по рядам (перечни возможностей) или по столбцам (ACL-списки).

Защищенные системы можно разработать, но безопасность должна быть главной целью с самого начала работы над проектом операционной системы. Вероятно, наиболее важное правило разработки системы заключается в создании минимальной высоконадежной вычислительной базы TCB, которую нельзя обойти, пытаясь получить доступ к любому ресурсу. Многоуровневая система безопасности может основываться на модели БеллаЛа–Падулы, разработанной для хранения секретов, или на модели Биба, созданной для поддержания целостности системы. В Оранжевой книге описываются требования, которым должны удовлетворять надежные системы. Наконец, даже если система с большой вероятностью является надежной, следует уделить внимание тайным каналам, которые легко могут низвергнуть систему, если ее модель не учитывает возможности их наличия.

Вопросы

1. Рассмотрим шифр с секретным ключом, представляющий собой матрицу 26×26, заголовки столбцов и строк которой содержат символы $ABC...Z$. Открытый текст шифруется по два символа. Первый символ указывает столбец, а второй — строку. В ячейке матрицы на пересечении столбца и строки содержатся два символа зашифрованного текста. Какие правила должны выполняться для этой матрицы? Сколько возможно ключей у такого шифра?
2. Расшифруйте следующее сообщение, составленное с помощью моноалфавитного шифра. Открытый текст, состоящий только из букв, представляет собой хорошо известный отрывок из поэмы Льюиса Кэрролла.

```
kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm
ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm
sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk
```

3. Рассмотрите следующий способ шифрования файла. Для алгоритма шифрования используется два n-байтовых массива, A и B. Первые n байтов считываются из файла в массив A. Затем $A[0]$ копируется в $B[i]$, $A[1]$ копируется в $B[j]$, $A[2]$ копируется в $B[k]$ и т. д. После того как все n байтов скопированы в массив B, этот массив записывается в выходной файл, после чего в массив A считываются следующие n байтов. Эта процедура повторяется до тех пор, пока не будет зашифрован весь файл. Обратите внимание, что в данной схеме шифрования символы не заменяются одни другими, а меняются местами. Сколько ключей следует перебрать, чтобы путем полного перебора взломать данный шифр? Укажите преимущество данной схемы перед моноалфавитным подстановочным шифром.
4. Шифрование с секретным ключом более эффективно, нежели шифрование с открытым ключом, но оно требует, чтобы отправитель и получатель заранее договорились об используемом ключе. Предположим, что отправитель и получатель никогда не встречались, но существует доверенная третья сторона, у которой есть общий секретный ключ с отправителем и другой общий секретный ключ с получателем. Как при таких обстоятельствах отправитель и получатель могут установить новый общий секретный ключ?
5. Приведите простой пример математической функции, которая в первом приближении может использоваться как необратимая функция.
6. Отсутствие эха при вводе пароля безопаснее, чем вывод эха в виде звездочек вместо каждого символа, так как последний вариант позволяет определить длину пароля любому, кто окажется рядом с экраном. Если предположить, что пароль состоит только из букв в верхнем и нижнем регистре и цифр, а длина его — от 5 до 8 знаков, то насколько безопаснее вообще ничего не отображать?
7. Закончив учебное заведение, вы получаете место директора большого университетского компьютерного центра, который только что выгнал свой древний

мэйнфрейм на пастбище и переключился на большой сервер локальной сети с операционной системой UNIX. Вы начинаете работать. Через пятнадцать минут после начала работы ваша ассистентка вбегает в кабинет с криком: «Какие-то студенты обнаружили алгоритм, которым мы шифруем наши пароли, и поместили его в Интернете». Какие будут ваши действия?

8. Схема защиты Морриса—Томпсона с n-разрядными случайными числами («солью») была разработана, чтобы затруднить взломщику отгадывание паролей при помощи зашифрованного заранее словаря. Защищает ли такая схема от студентов, пытающихся угадать пароль суперпользователя? Предполагается, что файл паролей доступен для чтения.
9. Назовите три характеристики, которые должен иметь хороший биометрический индикатор, чтобы его можно было использовать для аутентификации при входе в систему.
10. Существует ли какой-либо способ использовать аппаратуру MMU для предотвращения атаки переполнением, показанную на рис. 9.6? Объясните, почему да, или почему нет.
11. У факультета технической кибернетики есть локальная сеть с большим количеством машин, работающих под управлением операционной системы UNIX. Пользователь на любой машине может ввести команду вида `machine4 who`, и эта команда будет выполнена на компьютере *machine4*, для чего пользователю не нужно регистрироваться на удаленном компьютере. Это свойство реализовано следующим образом. Ядро системы машины пользователя посылает команду и ее UID удаленной машине. Надежна ли такая схема, если ядрам системы можно доверять? Что, если одна из машин представляет собой персональный компьютер студента, на который не установлена защита?
12. Что общего у реализации паролей в операционной системе UNIX со схемой Лампорта для регистрации по незащищенной сети?
13. Схема одноразовых паролей Лампорта использует пароли в обратном порядке. Не проще ли было бы в первый раз использовать $f(s)$, во второй — $f(f(s))$ и т. д.?
14. По мере того как Интернет-кафе становятся все популярнее, все больше людей хотело бы иметь возможность доступа к своей корпоративной сети из любого Интернет-кафе мира. Опишите способ формирования подписанного документа, отправляемого из Интернет-кафе с помощью смарт-карты (предполагается, что все компьютеры в Интернет-кафе оборудованы устройствами чтения смарт-карт). Безопасна ли такая схема?
15. Возможна ли атака с использованием троянского коня в системе, защищенной перечнями возможностей?
16. Назовите свойство компилятора С, способное устранить большое количество дыр в системе безопасности. Почему оно пока не получило более широкого применения?
17. При удалении файла его блоки, как правило, возвращаются в список свободных блоков, но их содержимое не стирается. Как вы полагаете, будет ли хорошей идеей, если операционная система будет очищать каждый блок

перед тем, как его освободит? Рассмотрите в вашем ответе факторы безопасности и производительности, а также покажите, какой эффект окажет данная схема на каждый фактор.

18. Как можно модифицировать операционную систему TENEX, чтобы избежать проблемы с паролями, описанной в тексте?
19. Как может паразитический вирус
    а) гарантировать, что он будет выполнен, прежде чем выполнится зараженная им программа;
    б) и вернуть управление зараженной программе после того, как он выполнит свои функции?
20. Некоторые операционные системы требуют, чтобы разделы диска начинались в начале дорожки диска. Как это облегчает жизнь вирусу, заражающему загрузочный сектор?
21. Измените программу в листинге 9.4 так, чтобы она находила вместо исполняемых файлов все программы на языке C.
22. Вирус на рис. 9.10, *г* зашифрован. Как может аналитик из антивирусной лаборатории определить, какая часть файла представляет собой ключ к шифру, чтобы расшифровать его и восстановить исходный текст вируса? Что может сделать автор вируса для усложнения этой задачи?
23. Вирус на рис. 9.10, *в* содержит программы сжатия и распаковки. Распаковщик необходим для распаковки и запуска вируса. Для чего нужен запаковщик?
24. Назовите один недостаток полиморфного шифрующегося вируса *с точки зрения автора вируса*.
25. Часто можно встретить следующую инструкцию для восстановления от вирусной атаки:
    а) загрузите зараженную систему;
    б) заархивируйте все файлы на внешний носитель;
    в) отформатируйте диск программой *fdisk*;
    г) переустановите операционную систему с оригинального CD-ROM;
    д) перезагрузите файлы с внешнего носителя.

    Назовите две серьезные ошибки данной инструкции.
26. Могут ли в операционной системе UNIX существовать вирусы-компаньоны (вирусы, которые не модифицируют существующие файлы)? Если да, то как? Если нет, то почему?
27. В чем заключается различие между вирусом и червем? Как они размножаются?
28. Для распространения программ или программных обновлений часто используются самораспаковывающиеся архивы, содержащие один или несколько запакованных файлов и программу распаковки. Рассмотрите влияние применения данной техники на безопасность.

29. На некоторых машинах команда SHR, используемая в листинге 9.6, заполняет неиспользуемые биты нулями, на других машинах для этого используется знаковый бит. Имеет ли значение, какой тип команды сдвига используется для корректности программы в листинге 9.6? Если да, то которая команда лучше?

30. Представьте информацию о владельцах и разрешениях, показанную в листинге каталога операционной системы UNIX, в виде матрицы защиты. *Примечание*: пользователь *asw* является членом сразу двух групп: *users* и *devel*, а пользователь *gmw* является членом только одной группы *users*. Оба пользователя и обе группы следует представить в виде доменов, так что матрица должна иметь четыре строки (по одной для каждого домена) и четыре столбца (по одной для каждого файла).

```
-rw-r--r-- 2  gmw  users  908    May 26 16:45  PPP-Notes
-rwxr-xr-x 1  asw  devel  432    May 13 12:35  progl
-rw-rw---- 1  asw  users  50094  May 30 17:51  project.t
-rw-r----- 1  asw  devel  13124  May 31 14:30  splash.gif
```

31. Представьте информацию о владельцах и разрешениях, показанную в листинге каталога в предыдущей задаче, в виде ACL-списка.

32. Модифицируйте ACL-список для одного файла, чтобы предоставить или запретить доступ, который не может быть выражен с помощью системы *rwx*, использующейся в UNIX. Объясните сделанные изменения.

33. Чтобы обеспечить возможность проверки того, что апплет был подписан доверенным производителем, производитель апплета может включить в него сертификат, подписанный доверенной третьей стороной, содержащий открытый ключ. Однако, чтобы прочитать сертификат, пользователю нужен открытый ключ доверенной третьей стороны. Этот ключ может быть подтвержден четвертой доверенной стороной, но тогда пользователю понадобится и этот открытый ключ. Похоже, что конца у этого метода не видно, тем не менее существующие браузеры пользуются им. Как это работает?

34. В матрице, управляющей доступом, ряды означают домены, а колонки — объекты. Что произойдет, если один объект нужен сразу в двух доменах?

35. В данной главе обсуждались два различных механизма защиты: перечни возможностей и списки управления доступом. Какой из этих двух механизмов может быть применен для каждой из следующих проблем:
    а) Кен хочет, чтобы его файлы могли читать все, кроме его напарника по офису;
    б) Митч и Стив хотят вместе пользоваться некоторыми секретными файлами;
    в) Линда хочет сделать открытыми некоторые из своих файлов.

36. В схеме перечней возможностей системы Amoeba пользователь может попросить сервер создать новый элемент перечня возможностей с меньшими правами, который затем он может передать своему другу. Что произойдет, если друг попросит сервер еще более сократить права, чтобы он смог передать новый элемент перечня возможностей кому-то еще?

37. На рис. 9.22 от процесса *B* к объекту *1* нет стрелки. Будет ли допустима такая стрелка? Если нет, то какому правилу она противоречит?
38. На рис. 9.22 нет стрелки от объекта *2* к процессу *A*. Будет ли допустима такая стрелка? Если нет, то какому правилу она противоречит?
39. Если бы в схеме на рис. 9.22 были разрешены сообщения между процессами, какие правила следовало бы для них установить? В частности, для процесса *B*: каким процессам мог бы он посылать сообщения, а каким нет?
40. Рассмотрите стеганографическую систему на рис. 9.25. Каждый пиксел может быть представлен в пространстве цветов как точка в 3-мерной системе с осями R, G и B. Используя это пространство, объясните, что происходит с цветовым разрешением при использовании стеганографии.
41. Текст на нормальном человеческом языке в формате ASCII с помощью различных алгоритмов сжатия может быть сжат, по меньшей мере, на 50 %. Учитывая это, сколько текста в формате ASCII вместит в себя графическое изображение размером 1600 × 1200 пикселов, если использовать все тот же метод стеганографии? Насколько увеличится размер изображения в результате применения данного метода (предполагается, что шифрование не применяется, или шифрование не увеличивает размеров данных)? Чему равна эффективность данной схемы, то есть отношение полезной нагрузки к общему числу передаваемых байтов?
42. Предположим, что тесно связанная группа политических диссидентов, живущих в государстве с репрессивным режимом, использует стеганографию, чтобы посылать во внешний мир сообщения о положении в их стране. Правительство знает об этом и борется с группой, посылая поддельные изображения, содержащие фальшивые стеганографические сообщения. Как могут диссиденты попытаться помочь людям отличить настоящие сообщения от фальшивых?
43. Напишите пару сценариев оболочки для отправки и получения текстового сообщения по тайному каналу в операционной системе UNIX. (*Подсказка*: используйте для тайного сигнала время выполнения процесса. Команда *sleep* гарантированно работает в течение минимального времени, указанного в качестве ее аргумента, а команда *ps* может использоваться для просмотра всех процессов.)
44. Напишите пару программ на C или на языке сценариев оболочки для отправки и получения сообщения по тайному каналу в операционной системе UNIX. *Подсказка*: бит разрешения файла может быть виден, даже если любой доступ к файлу запрещен, а команда или системный вызов *sleep* гарантированно создает задержку на определенный интервал времени, указанного в качестве аргумента. Измерьте скорость передачи данных в простаивающей системе. Затем искусственно создайте значительную нагрузку, запустив множество фоновых процессов, и снова измерьте скорость передачи данных.

Глава 10
Рассмотрение конкретных случаев: UNIX и Linux

В предыдущих главах мы познакомились с принципами многих операционных систем, с абстракциями, алгоритмами и общими методами. Теперь настало время взглянуть на некоторые конкретные системы, чтобы увидеть, как эти принципы применяются в реальном мире. Мы начнем наше изучение примеров с операционной системы UNIX, так как она работает на бо́льших типах компьютеров, чем любая другая операционная система. Система UNIX доминирует на рабочих станциях старших моделей и серверах, но она также используется на различных системах, от ноутбуков до суперкомпьютеров. Разработка этой операционной системы была подчинена строгому следованию определенной цели, и, несмотря на свой возраст, она до сих пор современна и элегантна. Система UNIX иллюстрирует множество важных принципов построения операционных систем, многие из которых были позаимствованы другими операционными системами.

Наше обсуждение системы UNIX мы начнем с ее истории и пути развития. Затем будет предоставлен общий обзор системы, который должен будет дать представление о том, как она работает. Этот обзор особенно важен для читателей, знакомых только с системой Windows, так как последняя скрывает от пользователя практически все детали системы. Хотя графические интерфейсы могут быть крайне удобными для начинающих пользователей, они обладают низкой гибкостью и не дают представления о том, как работает система.

Затем мы подойдем к сердцу этой главы, к изучению процессов управления памятью, ввода-вывода, файловой системы и безопасности в системе UNIX. Для каждой темы мы сначала обсудим фундаментальные понятия, затем системные вызовы и, наконец, методы реализации.

Одна из проблем, с которой мы столкнемся, заключается в том, что существует множество клонов и версий системы UNIX, включая AIX, BSD, 1BSD, HP-UX, Linux, MINIX, OSF/1, SCO UNIX, System V, Solaris, XENIX, а также многие другие, к тому же каждая из них распадается на свои подверсии. К счастью, фундаментальные принципы и системные вызовы практически для всех этих систем во многом совпадают. Более того, сходными являются общие стратегии реализации, алгоритмы и структуры данных, хотя имеются некоторые различия. В данной главе при обсуждении реализации будет приведено несколько примеров, в первую очередь системы 4.4BSD (составляющей основу для FreeBSD), System V Release 4,

и Linux. Дополнительные сведения о различных вариантах системы UNIX можно найти в [22, 132, 227, 230, 252, 334].

История UNIX

У операционной системы UNIX долгая и очень интересная история, поэтому с нее мы и начнем наше знакомство с этой системой. То, что началось как развлечение одного молодого исследователя, стало индустрией с оборотом во много миллионов долларов, в которую включились университеты, многонациональные корпорации, правительства и международные организации. На следующих страницах мы рассмотрим, как разворачивалась эта история.

UNICS

В 40-е и 50-е годы все компьютеры были персональными компьютерами, по крайней мере, в следующем смысле: в те времена обычное использование компьютера заключалось в том, что пользователь записывался на определенный час, и вся машина на этот период оказывалась в его распоряжении. Физические размеры этих компьютеров были огромными, но работать на таком компьютере в каждый момент времени мог тогда только один пользователь (программист). Когда на смену этим машинам в 60-е годы пришли пакетные системы, программист стал приносить в машинный зал задание в виде колоды перфокарт. Если накапливалось достаточное количество заданий, оператор копировал их на магнитную ленту в единый пакет. От пробивки перфокарт до получения программистом распечатки проходило около часа или более. При такой схеме на отладку программ уходило очень много времени, так как всего одна не там набитая запятая могла привести к потере программистом нескольких часов.

Чтобы как-то усовершенствовать существовавшую схему, которую практически все считали неудовлетворительной и непродуктивной, в Дартмутском колледже и Массачусетсском технологическом институте были изобретены системы разделения времени. Дартмутская система, в которой работал только BASIC, имела кратковременный коммерческий успех, после чего она полностью исчезла. Система Массачусетсского технологического института, CTSS, была универсальной системой, получившей колоссальный успех в научных кругах. За короткое время исследователи из Массачусетсского технологического института объединили усилия с лабораторией Bell Labs и корпорацией General Electric (в те времена General Electric производили компьютеры) и начали разработку системы второго поколения **MULTICS** (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), уже обсуждавшейся в главе 1.

Хотя лаборатория Bell Labs была одним из основополагающих партнеров проекта MULTICS, она вскоре вышла из него, в результате чего один из исследователей этой лаборатории, Кен Томпсон, оказался в ситуации поиска новых интересных дел. В конце концов, он решил сам написать (на ассемблере) усеченный вариант системы MULTICS, для чего у него был списанный мини-компьютер PDP-7. Несмотря на крошечные размеры PDP-7, система Томпсона работала и позволяла

Томпсону продолжать разработки новой операционной системы. Впоследствии еще один исследователь лаборатории Bell Labs, Брайан Керниган, как-то в шутку назвал эту систему **UNICS** (UNiplexed Information and Computing Service — примитивная информационная и вычислительная служба). Несмотря на все каламбуры и шутки на тему о кастрированной системе MULTICS (кое-кто предлагал назвать систему Томпсона EUNUCHS, то есть евнухом), кличка, данная Керниганом, прочно пристала к новой системе, хотя написание этого слова стало слегка короче, при том же произношении, превратившись в **UNIX**.

PDP-11 UNIX

Работа Томпсона произвела на его коллег из лаборатории Bell Labs столь сильное впечатление, что вскоре к нему присоединился Деннис Ритчи, а чуть позднее и весь его отдел. На это время приходятся два технологических усовершенствования. Во-первых, система UNIX была перенесена с устаревшей машины PDP-7 на более современные компьютеры PDP-11/20, а позднее на PDP-11/45 и PDP-11/70. Последние две машины доминировали в мире мини-компьютеров в течение большей части 70-х годов. Компьютеры PDP-11/45 и PDP-11/70 представляли собой мощные по тем временам машины с большой физической памятью (256 Кбайт и 2 Мбайт соответственно). Кроме того, они обладали аппаратной защитой памяти, что позволяло поддерживать одновременную работу нескольких пользователей. Однако это были 16-разрядные машины, и это ограничивало адресное пространство процессов 64 Кбайт для команд и 64 Кбайт для данных, несмотря на то, что у этих машин физической памяти было значительно больше 64 Кбайт[1].

Второе усовершенствование касалось языка, на котором писалась операционная система UNIX. Уже давно стало очевидно, что необходимость переписывать всю систему заново для каждой новой машины — занятие отнюдь не веселое, поэтому Томпсон решил переписать UNIX на языке высокого уровня, который он сам специально разработал и назвал языком **B**. Язык B представлял собой упрощенную форму языка BCPL (который, в свою очередь, был упрощенным языком CPL, подобно PL/1, никогда не работавшем). Эта попытка оказалась неудачной из-за слабостей языка B, в первую очередь, из-за отсутствия в нем структур данных. Тогда Ритчи разработал следующий язык, явившийся преемником языка B, который, естественно, получил название C, и написал для него прекрасный компилятор. Вместе Томпсон и Ритчи переписали UNIX на C. Язык C оказался как раз тем языком, который и был нужен в то время, и он сохраняет лидирующие позиции в области системного программирования до сих пор.

В 1974 г. Ритчи и Томпсон опубликовали ставшую важной вехой в истории компьютеров статью об операционной системе UNIX [275]. За работу, описанную в данной статье, им позднее ассоциацией по вычислительной технике ACM была присуждена престижная премия Тьюринга [274, 330]. Публикация этой статьи привела к тому, что многие университеты выстроились в очередь в лабораторию Bell Labs за копией системы UNIX. Корпорация AT&T, являвшаяся учредителем

[1] Точнее, у этих машин было одно общее 64-килобайтное адресное пространство и для команд, и для данных. Аналоги этих машин в СССР назывались СМ-4 и СМ-1420. — *Примеч. перев.*

лаборатории Bell Labs, была в то время регулируемой монополией и ей не разрешалось заниматься компьютерным бизнесом, поэтому она не возражала против того, чтобы университеты получали лицензии на право использования системы UNIX за умеренную плату.

По случайному стечению обстоятельств, которые часто формируют историю, машина PDP-11 использовалась на факультетах кибернетики практически каждого университета, а операционные системы, с которыми поставлялись эти компьютеры, профессоры и студенты считали ужасными. Операционная система UNIX быстро заполнила имевшийся вакуум, не в последнюю очередь также благодаря тому, что система поставлялась с полным комплектом исходных текстов, поэтому новые владельцы системы могли без конца подправлять и совершенствовать ее. Операционной системе UNIX было посвящено множество научных симпозиумов, на них докладчики рассказывали о скрытых ошибках в ядре, которые им удалось обнаружить и исправить. Австралийский профессор Джон Лайонс написал к исходному тексту системы UNIX комментарий стилем, обычно использующимся в трактатах о Джеффри Чосере или Вильяме Шекспире [211]. Книга описывала систему UNIX Version 6, названную так потому, что эта версия операционной системы была описана в шестом издании руководства программиста UNIX Programmer's Manual. Исходный текст системы состоял всего из 8200 строк на С и 900 строк ассемблера. В результате новые идеи и усовершенствования системы распространялись с огромной скоростью.

Через несколько лет Version 6 сменила Version 7, которая стала первой переносимой версией операционной системы UNIX (она работала как на машинах PDP-11, так и на Interdata 8/32). Эта версия системы уже состояла из 18 800 строк на С и 2100 ассемблерных строк. На Version 7 выросло целое поколение студентов, которые, закончив свои учебные заведения и начав работу в промышленности, в значительной степени содействовали дальнейшему распространению системы UNIX. К середине 80-х операционная система UNIX широко применялась на мини-компьютерах и инженерных рабочих станциях самых различных производителей. Многие компании даже приобрели лицензии на исходные тексты, чтобы производить свои версии системы UNIX. Одной из таких компаний была небольшая начинающая фирма Microsoft, в течение нескольких лет продававшая Version 7 под именем XENIX, пока ее интересы не повернулись в другую сторону.

Переносимая система UNIX

После того как система UNIX была переписана на языке высокого уровня С, задача переноса ее на новые машины стала значительно более простым делом. Для переноса системы сначала требуется написать для новой машины компилятор с языка С. Затем для устройств ввода-вывода новой машины, таких как терминалы, принтеры и диски, нужно написать драйверы устройств. Хотя драйвер может быть написан и на С, его нельзя просто перекомпилировать на новой машине и запустить на ней, поскольку нет двух одинаково работающих дисков. Наконец, требуется переписать заново, как правило, на ассемблере, небольшое количество машинно-зависимого кода, например обработчики прерываний и процедуры управления памятью.

Первым компьютером, на который была перенесена с машины PDP-11 операционная система UNIX, был мини-компьютер Interdata 8/32. При этом выяснилось следующее: система UNIX неявно предполагала, что целые числа должны быть 16-разрядными, адреса также состоять из 16 бит (в результате чего максимальный размер программы ограничивался размером 64 Кбайт) и что у компьютера имеется ровно три регистра для хранения важных переменных. Ни одно из этих предположений не было справедливым для мини-компьютера Interdata 8/32, поэтому для того, чтобы сделать систему UNIX действительно переносимой, пришлось немало потрудиться.

Другая проблема заключалась в том, что хотя компилятор Ритчи был быстрым и формировал хороший объектный код, он мог создавать только объектный код PDP-11. Вместо того чтобы написать новый компилятор специально для Interdata 8/32, Стив Джонсон из все той же лаборатории Bell Labs разработал и реализовал **переносимый компилятор языка С**. Этот компилятор можно было настроить на создание объектного кода для практически любой машины, для чего требовался умеренный объем работ. В течение нескольких лет почти все компиляторы языка С, кроме компиляторов для машин PDP-11, основывались на компиляторе Джонсона, что значительно помогло распространению системы UNIX на новые компьютеры.

Процесс переноса операционной системы UNIX на мини-компьютер Interdata 8/32 шел медленно, так как вся работа должна была производиться на единственной в лаборатории машине, на которой работала система UNIX, на PDP-11. Однако случилось так, что компьютер PDP-11 оказался на пятом этаже лаборатории, тогда как Interdata 8/32 был установлен на первом этаже. Создание новой версии системы означало ее компиляцию на пятом этаже и запись на магнитную ленту, которая физически переносилась на первый этаж, чтобы посмотреть, работает ли она. Уже через несколько месяцев подобной работы у разработчиков появился сильный интерес к возможности соединения этих двух машин электронным способом. Вся работа с сетью в системе UNIX началась именно с соединения этих двух машин. После Interdata система UNIX была перенесена на VAX, а затем и на другие компьютеры.

После того как в 1984 году правительство США разделило корпорацию AT&T, компания получила законную возможность инвестировать средства в компьютерную промышленность, что она и сделала. Вскоре после этого компания AT&T выпустила на рынок первый коммерческий вариант системы UNIX, System III. Ее выход на рынок был не очень успешным, поэтому через год она была заменена улучшенной версией, System V. Что случилось с System IV, до сих пор остается одной из неразгаданных тайн компьютерного мира. Оригинальную систему System V впоследствии сменили выпуски 2, 3 и 4 все той же System V, каждый последующий выпуск более сложный и громоздкий, чем предшествующий. В процессе усовершенствований оригинальная идея, лежащая в основе системы UNIX и заключающаяся в простоте и элегантности системы, была в значительной мере утрачена. Хотя группа Ритчи и Томпсона позднее выпустила 8-ю, 9-ю и 10-ю редакции системы UNIX, они не получили широкого распространения, так как компания AT&T все свои усилия на рынке вкладывала в продажу версии System V. Однако некоторые идеи из 8-й, 9-й и 10-й редакции системы в конце концов были

включены в System V. Наконец, компания AT&T решила, что хочет быть телефонной компанией, а не компьютерной фирмой и в 1993 году продала весь свой бизнес, связанный с системой UNIX, корпорации Novell, которая, в свою очередь, в 1995 году перепродала его компании Santa Cruz Operation. К этому времени стало практически неважным, кому принадлежит этот бизнес, так как почти у всех основных компьютерных компаний уже были лицензии.

Berkeley UNIX

Калифорнийский университет в Беркли был одним из многих университетов, приобретших UNIX Version 6 практически с момента ее выхода. Поскольку с системой поставлялся полный комплект исходных текстов, университет мог существенно модифицировать систему. При финансовой поддержке управления перспективного планирования научно-исследовательских работ ARPA (Advanced Research Projects Agency) при Министерстве обороны США университет в Беркли разработал и выпустил улучшенную версию операционной системы UNIX для мини-компьютера PDP-11, названую **1BSD** (First Berkeley Software Distribution — программное изделие Калифорнийского университета, 1-я версия). Вслед за этой магнитной лентой вскоре появилась 2BSD, также для PDP-11.

Более важным событием был выпуск версии 3BSD и ее преемника 4BSD, уже рассчитанных на 32-разрядные машины VAX. Хотя компания AT&T распространяла свою собственную версию для VAX, называвшуюся **32V**, по существу, это была Version 7. В противоположность ей система 4BSD (включая 4.1BSD, 4.2BSD, 4.3BSD и 4.4BSD) содержала большое количество усовершенствований. Важнейшими из них были использование виртуальной памяти и страничная подкачка файлов, что позволяло создавать программы, большие по размеру, чем физическая память. Другое изменение заключалось в поддержке имен файлов длиной более 14 символов. Реализация файловой системы также была изменена, благодаря чему работа с файловой системой стала существенно быстрее. Более надежной стала обработка сигналов. В 4-й версии Berkeley UNIX появилась поддержка сетей, в результате чего используемый в 4BSD протокол **TCP/IP** стал стандартом де-факто в мире UNIX, а позднее и в Интернете, в котором преобладают серверы на базе системы UNIX.

Университет в Беркли также добавил значительное количество утилит для системы UNIX, включая новый редактор *vi* и новую оболочку *csh*, компиляторы языков Pascal и Lisp и многое другое. Все эти усовершенствования привели к тому, что многие производители компьютеров (Sun Microsystems, DEC и другие) стали основывать свои версии системы UNIX на Berkeley UNIX, а не на «официальной» версии компании AT&T, System V. В результате Berkeley UNIX получила широкое распространение в академических и исследовательских кругах. Дополнительные сведения о системе Berkeley UNIX см. [230].

Стандартная система UNIX

К концу 80-х широкое распространение получили две различные и в чем-то несовместимые версии системы UNIX: 4.3BSD и System V Release 3. Кроме того, практически каждый производитель добавлял свои нестандартные усовершенство-

вания. Этот раскол в мире UNIX, вместе с тем фактом, что стандарта на формат двоичных программ не было, сильно замедлил коммерческое признание операционной системы UNIX. Производители программного обеспечения не могли написать пакет программ для системы UNIX так, чтобы он гарантированно мог быть запущен на любой системе UNIX (как, например, это делалось в системе MS-DOS). Различные попытки стандартизации системы UNIX провалились с самого начала. Например, корпорация AT&T выпустила стандарт **SVID** (System V Interface Definition — описание интерфейса UNIX System V), в котором определялись все системные вызовы, форматы файлов и т. д. Этот документ был попыткой построить в одну шеренгу всех производителей UNIX System V, но он не оказал никакого влияния на вражеский лагерь (BSD), который просто проигнорировал его.

Первая попытка примирить два варианта системы UNIX была предпринята при содействии Совета по стандартам при Институте инженеров по электротехнике и электронике IEEE Standard Boards, глубокоуважаемой и, что еще важнее, нейтральной организации. В этой работе приняли участие сотни людей из промышленности, академических и правительственных организаций. Коллективное название данного проекта — **POSIX**. Первые три буквы этого сокращения означали Portable Operating System — переносимая операционная система. Буквы *IX* в конце слова были добавлены, чтобы имя проекта выглядело юниксообразно.

После большого количества высказанных аргументов и контраргументов, опровержений и опровергнутых опровержений, комитет POSIX выработал стандарт, известный как **1003.1**. Этот стандарт определяет набор библиотечных процедур, которые должна предоставлять каждая соответствующая данному стандарту система UNIX. Большая часть этих процедур обращается к системному вызову, но некоторые из них могут быть реализованы вне ядра. Типичными процедурами являются *open*, *read* и *fork*. Идея стандарта POSIX заключается в том, что производитель программного обеспечения при написании программы использует только процедуры, описанные в стандарте 1003.1, таким образом, гарантируя, что эта программа будет работать на любой версии системы UNIX, поддерживающей данный стандарт.

Хотя большинство комитетов по стандартам, как правило, создают нечто ужасное, сплошь состоящее из компромиссов, стандарт 1003.1 заметно отличается от общего правила в лучшую сторону, особенно если учитывать большое число заинтересованных сторон, принимавших участие в его разработке. Вместо того чтобы принять за точку отсчета объединение множеств всех свойств System V и BSD (норма для большинства комитетов по стандартам), комитет IEEE взял за основу пересечение множеств. То есть в первом приближении дело обстоит так: если какое-либо свойство присутствовало как в System V, так и в BSD, то оно включалось в стандарт. В противном случае это свойство в стандарт не включалось. В результате применения такого алгоритма стандарт 1003.1 сильно напоминает прямого общего предка систем System V и BSD, а именно Version 7. От Version 7 стандарт сильнее всего отличается в двух областях: обработке сигналов (что по большей части взято из BSD) и управлению терминалом, что представляет собой нововведение. Документ 1003.1 написан так, чтобы как разработчики операционной системы, так и создатели программного обеспечения были способны его понять, что также ново в мире стандартов, хотя в настоящее время уже полным ходом ведется работа по исправлению этого нестандартного для стандартов свойства.

Стандарт 1003.1 описывает только системные вызовы, принят также ряд документов, стандартизирующих потоки, утилиты, сетевое программное обеспечение и многие другие особенности системы UNIX. Кроме того, язык С также был стандартизирован Национальным институтом стандартизации США ANSI и Международной организацией по стандартизации ISO.

К сожалению, после успешного принятия стандарта, объединившего System V и BSD, в мире UNIX снова произошел раскол. Группе производителей компьютеров и программного обеспечения, среди которых были такие фирмы, как IBM, DEC, Hewlett-Packard, не понравилось, что корпорация AT&T владеет остальной частью системы UNIX. Поэтому они создали консорциум, названный **OSF** (Open Software Foundation — Фонд открытого программного обеспечения), чтобы создать систему, удовлетворяющую всем стандартам IEEE и другим стандартам, но также содержащую множество дополнительных свойств. Среди этих свойств оконная система (X11), графический интерфейс пользователя (MOTIF), распределенные вычисления (DCE, Distributed Computing Environment — среда распределенных вычислений), распределенное управление (DME, Distributed Management Environment — среда распределенного управления), а также многое другое.

В ответ на это корпорация AT&T создала собственный консорциум **UI** (UNIX International), чтобы заниматься практически тем же самым. Версия UI системы UNIX основывалась на System V. В результате в мире оказалось две мощные промышленные группы, каждая из которых предлагала пользователю свою версию системы UNIX, так что пользователи нисколько не приблизились к единому стандарту. Однако рынок решил, что System V лучше, чем OSF, поэтому второй вариант системы постепенно исчез из употребления. У некоторых компаний сейчас есть свои версии системы UNIX, как, например, система Solaris корпорации Sun (основанная на System V).

MINIX

У всех этих систем есть общее свойство: все они большие и сложные, что в определенном смысле противоречит оригинальной идее, лежавшей в основе системы UNIX. Даже если бы все исходные тексты систем продолжали свободно распространяться, что в большинстве случаев уже давно не так, все равно одному человеку просто не под силу понять их. Эта ситуация привела к тому, что автор этой книги написал новую юниксоподобную систему, достаточно небольшую, чтобы ее было можно понять, с доступным полным исходным текстом, для использования в учебных целях. Эта система состояла из 11 800 строк на С и 800 строк на ассемблере. Она была выпущена в 1987 году и функционально практически эквивалентна системе Version 7 UNIX, бывшей оплотом большинства факультетов кибернетики в эпоху PDP-11.

Система MINIX была одной из первых юниксообразных систем, основанной на схеме микроядра. Идея микроядра заключается в том, чтобы реализовать как можно меньше функций в ядре, в результате чего можно создать надежное и эффективное ядро. Соответственно, задачи управления памятью и файловой системой были перемещены в процессы пользователя. Ядро в основном обрабатывало передачу сообщений между процессами, почти не занимаясь другими задачами.

Ядро состояло из 1600 строк на С и 800 ассемблерных строк. По техническим причинам, связанным с архитектурой процессора Intel 8088, драйверы устройств ввода-вывода (еще 2900 строк на С) также были размещены в ядре. Файловая система (5100 строк на С) и менеджер памяти (2200 строк на С) работали как два раздельных пользовательских процесса.

Преимущество микроядер перед монолитными системами заключается в том, что устройство микроядра легко понять, да и поддержка системы, основанной на микроядре, проще благодаря модульной структуре такой системы. Кроме того, перемещение программного обеспечения из ядра в пространство пользователя существенно повышает надежность системы, так как сбой процесса, работающего в режиме пользователя, способен нанести меньший ущерб, чем сбой компонента в режиме ядра. Основной недостаток состоит в несколько меньшей производительности, связанной с дополнительными переключениями из режима пользователя в режим ядра. Однако производительность — не единственное достоинство системы. На всех современных системах UNIX оконная система X Windows работает в режиме пользователя, в результате чего производительность несколько снижается, зато достигается большая модульность (в отличие от системы Windows, у которой весь графический интерфейс пользователя расположен в ядре). Среди других хорошо известных примеров схемы микроядра того времени можно назвать Mach [4] и Chorus [282]. Обсуждение производительности микроядерной системы UNIX приводится в [42].

Уже через несколько месяцев после своего появления система MINIX стала чем-то вроде объекта культа — у нее есть своя конференция, *comp.os.minix*, и более 40 000 пользователей. Многие пользователи сами стали писать команды и пользовательские программы, так что система MINIX стала продуктом коллективного творчества большого количества пользователей, обменивающихся своими разработками по Интернету. Она стала прототипом других коллективных работ, появившихся позднее. В 1997 году была выпущена версия 2.0 системы MINIX. Базовая система теперь включала в себя сетевое программное обеспечение, и ее размер вырос до 62 200 строк. О системе MINIX написана книга, содержащая 500 страниц исходного текста в приложении к книге, а также на поставляемом с книгой CD-ROM [326]. Исходный текст операционной системы MINIX можно бесплатно получить на web-сайте www.cs.vu.nl/~ast/minix.html.

Linux

В ранние годы развития системы MINIX и обсуждений этой системы в Интернете многие люди просили (а часто требовали) все больше новых и более сложных функций, и на эти просьбы автор часто отвечал отказом (сохраняя небольшие размеры системы, чтобы студенты могли полностью понять ее за один семестр). Эти отказы раздражали многих пользователей. В те времена бесплатной системы FreeBSD еще не было. Наконец через несколько лет финский студент Линус Торвальдс решил сам написать еще один клон системы UNIX, который он назвал **Linux**. Это должна была быть полноценная операционная система, со многими функциями, отсутствующими (по намерению авторов) в системе MINIX. Первая версия операционной системы Linux 0.01 была выпущена в 1991 году. Она была разработана

и собрана в системе MINIX и заимствовала некоторые идеи системы MINIX, начиная со структуры дерева исходных текстов и кончая структурой файловой системы. Однако, в отличие от микроядерной системы MINIX, Linux была монолитной системой, то есть вся операционная система помещалась в ядре. Размер исходного текста составил 9300 строк на С и 950 строк на ассемблере, что приблизительно совпадало с версией MINIX. Функционально первая версия Linux также практически почти не отличалась от MINIX.

Операционная система Linux быстро росла в размерах и впоследствии развилась в полноценный клон UNIX с виртуальной памятью, более сложной файловой системой и многими другими добавленными функциями. Хотя изначально система Linux работала только на процессоре Intel 386 (и даже содержала ассемблерный код 386-й машины посреди процедур на языке С), она была довольно быстро перенесена на другие платформы и теперь работает на широком спектре машин, как и UNIX. Однако одно из основных отличий системы Linux от других клонов системы UNIX заключается в использовании многих специальных особенностей компилятора *gcc*, поэтому, чтобы откомпилировать ее стандартным ANSI С компилятором, потребуется приложить немало усилий.

Следующим основным выпуском системы Linux была версия 1.0, появившаяся в 1994 году. Она состояла из 165 000 строк кода и включала новую файловую систему, отображение файлов на адресное пространство памяти и совместимое с BSD сетевое программное обеспечение с сокетами и TCP/IP. Она также включала многие новые драйверы устройств. Следующие два года выходили версии с незначительными исправлениями.

К этому времени операционная система Linux стала достаточно совместимой с UNIX, поэтому в нее было перенесено большое количество программного обеспечения UNIX, что значительно увеличило полезность рассматриваемой системы. Кроме того, операционная система Linux привлекла большое количество людей, которые начали работу над ее совершенствованием и расширением под общим руководством Торвальдса.

Следующий главный выпуск, версия 2.0, вышел в свет в 1996 году. Эта версия системы Linux состояла из 470 000 строк на С и 8000 строк ассемблерного текста. Она включала в себя поддержку 64-разрядной архитектуры, симметричной многозадачности, новых сетевых протоколов и прочих многочисленных функций. Значительную часть от общей массы исходного текста составляла внушительная коллекция различных драйверов устройств. Следом за версией 2.0 довольно часто выходили дополнительные выпуски.

В систему Linux была перенесена внушительная часть стандартного программного обеспечения UNIX, включая более 1000 утилит, оконную систему X Windows и большую часть сетевого программного обеспечения. Кроме того, специально для Linux было написано два различных графических интерфейса пользователя: GNOME и KDE. В общем, система Linux выросла в полноценный клон UNIX со всеми погремушками, какие только могут понадобиться фанату UNIX.

Необычной особенностью Linux является ее бизнес-модель: это свободно распространяющееся программное обеспечение. Ее можно скачать с различных Интернет-сайтов, например www.kernel.org. Система Linux поставляется вместе с лицензией, разработанной Ричардом Столманом, основателем Фонда бесплатно распростра-

няемых программ Free Software Foundation. Несмотря на то что система Linux распространяется бесплатно, эта лицензия, называющаяся **GPL** (GNU Public License — общедоступная лицензия), по длине превышает лицензию корпорации Microsoft для операционной системы Windows 2000. Она содержит сведения о том, что вы можете и чего не можете делать с исходным текстом. Пользователи могут свободно использовать, копировать, модифицировать и распространять дальше исходные тексты и двоичные файлы. Основной запрет касается отдельной продажи или распространения двоичного кода без исходных текстов. Исходные тексты должны либо поставляться вместе с двоичными файлами, либо предоставляться по требованию.

Хотя Торвальдс до сих пор довольно внимательно контролирует ядро системы, большое количество программ пользовательского уровня было написано другими многочисленными программистами, многие из которых изначально перешли на Linux из сетевых сообществ MINIX, BSD и GNU (Free Software Foundation). Однако по мере развития системы Linux все меньшая часть сообщества Linux желает ковыряться в исходном тексте (свидетельством тому служат сотни книг, описывающих, как установить систему Linux и как ею пользоваться, и только несколько книг, в которых обсуждается сама система, или как она работает). Кроме того, многие пользователи Linux теперь предпочитают бесплатному скачиванию системы по Интернету покупку одного из CD-ROM, распространяемых многочисленными коммерческими компаниями. На web-сайте www.linux.org перечислено более 50 компаний, продающих различные пакеты Linux. По мере того как все больше и больше компаний, занимающихся программным обеспечением, начинают продавать свои версии Linux, а все большее число производителей компьютеров поставляют систему Linux со своими машинами, граница между коммерческим и бесплатным программным обеспечением слегка размывается.

Интересно отметить, что когда мода на Linux начала набирать обороты, система получила поддержку с неожиданной стороны — от корпорации AT&T. В 1992 году университет в Беркли, лишившись финансирования, решил прекратить разработку BSD UNIX с последней версией 4.4BSD (которая впоследствии послужила основой для FreeBSD). Поскольку эта версия не содержала по существу кода AT&T, университет в Беркли выпустил это программное обеспечение под лицензией открытого исходного кода, которая позволяла всем делать все, что угодно, кроме одной вещи — подавать в суд на университет Калифорнии в Беркли. Корпорация AT&T, продолжающая контролировать систему UNIX в дополнение к своим основным занятиям, немедленно отреагировала — можете догадаться, как — подав в суд на университет Калифорнии в Беркли. Она мгновенно подала иск против компании BSDI, созданной разработчиками BSD UNIX для упаковки системы и поддержки продаж (примерно так сейчас поступают компании типа Red Hat с операционной системой Linux). Поскольку практически программы AT&T не использовались, судебное дело основывалось на нарушении авторского права и торговой марки, включая такие моменты, как телефонный номер компании BSDI 1-800-ITS-UNIX. Хотя этот спор в конце концов удалось урегулировать без судебного разбирательства, это не позволило выпустить на рынок FreeBSD в течение периода, достаточно долгого для того, чтобы система Linux успела развиться и упрочить свои позиции.

Если бы корпорация AT&T не подала судебного иска, то уже где-то году в 1993 началась бы серьезная борьба между двумя бесплатными версиями системы UNIX, распространяющимися с исходными текстами: царствующим чемпионом, системой BSD, матерой и устойчивой системой с многочисленными приверженцами в академической среде еще с 1977 года, и яростным молодым претендентом, системой Linux, всего лишь двух лет отроду, но уже с растущим числом сторонников среди индивидуальных пользователей. Кто знает, чем бы обернулась эта схватка двух бесплатных версий системы UNIX?

Если учитывать эту историю, строгое соответствие стандарту POSIX, а также пересечение сообществ пользователей, то не должно показаться удивительным, что многие черты системы Linux, системные вызовы, программы, библиотеки, алгоритмы и внутренние структуры данных очень схожи со своими аналогами в UNIX. Например, более 80 % от приблизительно 150 системных вызовов Linux представляют собой точные копии соответствующих системных вызовов в POSIX, BSD или System V. Таким образом, в первом приближении большая часть описания системы UNIX, приведенного в данной главе, также применимо и к Linux. В тех местах, где UNIX и Linux существенно различаются (например, в алгоритме планирования), это будет специально отмечено, и мы расскажем об обоих вариантах. Там же, где серьезных отличий не будет, для краткости мы будем просто описывать операционную систему UNIX. Следует предупредить читателя, что операционная система Linux стремительно развивается и тысячи программистов работают над ее совершенствованием, поэтому некоторая часть данного материала (основанная на версии 2.2), без всякого сомнения, довольно скоро устареет.

Обзор системы UNIX

В этом разделе будет представлено общее введение в операционную систему UNIX, а также описано, как ей пользоваться, для читателей, еще не знакомых с этой системой. Хотя разные версии системы UNIX различаются в мелких деталях, приведенный здесь материал применим ко всем версиям системы. Основное внимание в данном разделе будет уделено тому, как система UNIX выглядит на терминале. Последующие разделы будут посвящены системным вызовом и внутреннему устройству системы.

Задачи UNIX

Операционная система UNIX представляет собой интерактивную систему, разработанную для одновременной поддержки нескольких процессов и нескольких пользователей. Она была разработана программистами и для программистов, чтобы использовать ее в окружении, в котором большинство пользователей являются относительно опытными и занимаются проектами (часто довольно сложными) разработки программного обеспечения. Во многих случаях большое количество программистов активно сотрудничают в деле создания единой системы, поэтому в операционной системе UNIX есть достаточное количество средств, позволяющих программистам работать вместе и управлять совместным использованием общей информации. Очевидно, что модель группы опытных программистов, совместно

работающих над созданием передового программного обеспечения, существенно отличается от модели одиночного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражено в операционной системе UNIX от начала до конца.

Чего хотят от операционной системы хорошие программисты? Во-первых, большинство хотело бы, чтобы их система была простой, элегантной и непротиворечивой. Например, на нижнем уровне файл должен представлять собой просто набор байтов. Наличие различных классов файлов для последовательного и произвольного доступа, доступа по ключу, удаленного доступа и т. д. (как это реализовано на мэйнфреймах) просто является помехой. А если команда

```
ls A*
```

означает вывод списка всех файлов, имя которых начинается с буквы «А», то команда

```
rm A*
```

должна означать удаление всех файлов, имя которых начинается с буквы «А», а не одного файла, имя которого состоит из буквы «А» и звездочки. Эта характеристика иногда называется *принципом наименьшей неожиданности.*

Другое свойство, которое, как правило, опытные программисты желают видеть в операционной системе, — это мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать бесконечным числом способов, чтобы приспособить их для конкретного приложения. Одно из основных правил системы UNIX заключается в том, что каждая программа должна выполнять всего одну функцию, но делать это хорошо. Таким образом, компиляторы не занимаются созданием листингов, так как другие программы могут лучше справиться с этой задачей.

Наконец, у большинства программистов сильная неприязнь к бесполезной избыточности. Зачем писать *copy*, когда достаточно *cp*? Чтобы получить список всех строк, содержащих строку «ard» из файла *f*, программист в операционной системе UNIX вводит команду

```
grep ard f
```

Противоположный подход состоит в том, что программист сначала запускает программу *grep* (без аргументов), после чего программа *grep* приветствует программиста фразой: «Здравствуйте, я *grep*. Я ищу символьные строки в файлах. Введите, пожалуйста, искомую строку». Получив строку, программа *grep* просит задать имя файла. Затем она спрашивает, есть ли еще какие-либо файлы. Наконец она выводит листинг итогового задания и спрашивает, все ли верно. Хотя такой тип пользовательского интерфейса, возможно, удобен для начинающих пользователей, он бесконечно раздражает опытных программистов. То, что им нужно — это слуга, а не нянька.

Интерфейсы системы UNIX

Операционную систему UNIX можно рассматривать в виде пирамиды (рис. 10.1). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, терминалов и других устройств. На голом

«железе» работает операционная система UNIX. Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам создавать процессы, файлы и прочие ресурсы, а также управлять ими.



Рис. 10.1. Уровни операционной системы UNIX

Программы обращаются к системным вызовам, помещая аргументы в регистры центрального процессора (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра и передачи управления операционной системе UNIX. Поскольку на языке C невозможно написать команду эмулированного прерывания, этим занимаются библиотечные функции, по одной на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из программ, написанных на C. Каждая такая процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания TRAP. Таким образом, чтобы обратиться к системному вызову read, программа на C должна вызвать библиотечную процедуру *read*. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Другими словами, стандарт POSIX определяет библиотечные процедуры, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В стандарте даже не упоминаются фактические системные вызовы.

Помимо операционной системы и библиотеки системных вызовов, все версии UNIX содержат большое количество стандартных программ, некоторые из них описываются стандартом POSIX 1003.2, тогда как другие могут различаться в разных версиях системы UNIX. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускаются пользователем с терминала.

Таким образом, мы можем говорить о трех интерфейсах в операционной системе UNIX: интерфейсе системных вызовов, интерфейсе библиотечных функций и интерфейсе, образованным набором стандартных обслуживающих программ.

Хотя именно последний интерфейс большинство пользователей считает системой UNIX, в действительности он не имеет практически никакого отношения к самой операционной системе и легко может быть заменен.

В некоторых версиях системы UNIX, например, этот ориентированный на ввод с клавиатуры интерфейс пользователя был заменен графическим интерфейсом пользователя, ориентированным на использование мыши, для чего не потребовалось никаких изменений в самой системе. Именно эта гибкость сделала систему UNIX столь популярной и позволила ей пережить многочисленные изменения технологии, лежащей в ее основе.

Оболочка UNIX

У многих версий системы UNIX имеется графический интерфейс пользователя, схожий с популярными интерфейсами, примененными на компьютере Macintosh и впоследствии в системе Windows. Однако истинные программисты до сих пор предпочитают интерфейс командной строки, называемый **оболочкой** (shell). Подобный интерфейс значительно быстрее в использовании, существенно мощнее, проще расширяется и не раздражает пользователя необходимостью постоянно хвататься за мышь. Ниже будет кратко описана оболочка Бурна (*sh*). С тех пор было написано много других оболочек (*ksh*, *bash* и т. д.). Хотя система UNIX полностью поддерживает графическое окружение (X Windows), даже в этом мире многие программисты просто создают множество консольных окон и действуют так, как если бы у них была дюжина алфавитно-цифровых терминалов, на каждом из которых работала бы оболочка.

Когда оболочка запускается, она инициализируется, а затем печатает на экране символ приглашения к вводу (обычно это знак доллара или процента) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удается найти, оболочка запускает его. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка снова печатает приглашение и ждет ввода следующей строки. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, — это способность ввода с терминала и вывода на терминал, а также возможность запускать другие программы.

У команд оболочки могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

```
cp src dest
```

запускает программу *cp* с двумя аргументами *src* и *dest*. Эта программа интерпретирует первый аргумент как имя существующего файла. Она копирует этот файл и называет его копию *dest*.

Не все аргументы обязательно должны быть именами файлов. В строке

```
head -20 file
```

первый аргумент *–20* велит программе *head* напечатать первые 20 строк файла *file* вместо принятых по умолчанию 10 строк. Аргументы, управляющие работой коман-

ды или указывающие дополнительные значения, называются **флагами** или ключами и по соглашению обозначаются знаком тире. Тире требуется, чтобы избежать двусмысленности. Так, например, команда

```
head 20 file
```

вполне законна. Она велит программе *head* напечатать первые 10 строк файла *20*, а затем первые 10 строк файла *file*. Большинство команд системы UNIX могут принимать несколько флагов и аргументов.

Чтобы было легче указывать группы файлов, оболочка принимает так называемые **волшебные символы**, иногда называемые также **джокерами**. Например, символ звездочки означает все возможные варианты текстовой строки, так что строка

```
ls *.c
```

велит программе *ls* вывести список всех файлов, имя которых оканчивается на *.c*. Если файлы *x.c*, *y.c* и *z.c* существуют, то данная команда эквивалентна команде

```
ls x.c y.c z.c
```

Другим джокером является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

```
ls [ape]*
```

велит программе *ls* вывести список всех файлов, имя которых начинается с символов «a», «p» или «e».

Программа вроде оболочки не должна открывать терминал, чтобы прочитать с него или вывести на него строку. Вместо этого запускаемые программы автоматически получают доступ к файлу, называемому стандартным устройством ввода (standard input), и к файлу, называемому стандартным устройством вывода (standard output), а также к файлу, называемому **standard error** (стандартное устройство для вывода сообщений об ошибках). По умолчанию всем трем устройствам соответствует терминал, то есть клавиатура для ввода и экран для вывода. Многие программы в системе UNIX читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда

```
sort
```

вызывает программу *sort*, читающую строки с терминала (пока пользователь не нажмет комбинацию клавиш CTRL+D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод также можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Разрешается их одновременное использование в одной командной строке. Например, команда

```
sort <in >out
```

заставляет программу *sort* взять в качестве входного файл *in* и направить вывод в файл *out*. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, все сообщения об ошибках будут печататься на экране. Программа, считы-

вающая данные со стандартного устройства ввода, выполняющая определенную обработку этих данных и записывающая результат в поток стандартного вывода, называется **фильтром**.

Рассмотрим следующую командную строку, состоящую из трех отдельных команд:

```
sort <in >temp; head -30 <temp; rm temp
```

Сначала запускается программа *sort*, которая принимает данные из файла *in* и записывает результат в файл *temp*. Когда она завершает свою работу, оболочка запускает программу *head*, веля ей распечатать первые 30 строк из файла *temp* на стандартном устройстве вывода, которым по умолчанию является терминал. Наконец, временный файл *temp* удаляется.

В системе UNIX часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном выше примере для этого использовался временный файл *temp*. Однако система UNIX предоставляет более простой способ для этого. В командной строке

```
sort <in | head -30
```

для этого используется вертикальная черта, называемая **символом канала**. Этот символ означает, что вывод программы *sort* должен использоваться в качестве входа для программы *head*, что позволяет обойтись без явного указания оболочке создать временный файл, а потом удалить его. Набор команд, соединенных символом канала, называется **конвейером** и может содержать произвольное количество команд. Пример четырехкомпонентного конвейера показан в следующей строке:

```
grep ter * .t | sort | head -20 | tail -5 >foo
```

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, оканчивающихся на *.t*, после чего они сортируются. Первые 20 строк выбираются программой *head*, которая передает их программе *tail*, записывающей последние пять строк (то есть строки с 16 по 20 в отсортированном списке) в файл *foo*. Вот пример того, как операционная система UNIX предоставляет основные строительные блоки (фильтры), каждый из которых выполняет определенную работу, вместе с механизмом, позволяющим объединять их практически неограниченными способами.

UNIX является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании символа амперсанда в конце строки. Таким образом, строка

```
wc -l <a >b &
```

запустит программу подсчета количества слов *wc*, которая сосчитает число строк (флаг *–l*) во входном файле *a* и запишет результат в файл *b*, но будет делать это в фоновом режиме. Как только команда введена пользователем, оболочка напечатает символ приглашения к вводу и перейдет в режим ожидания следующей команды. Конвейеры также могут выполняться в фоновом режиме, например:

```
sort <x | head &
```

Можно одновременно запустить несколько фоновых конвейеров.

Список команд оболочки может быть помещен в файл, а затем этот файл с командами может быть выполнен, для чего нужно запустить оболочку с этим файлом в качестве входного аргумента. Вторая программа оболочки просто выполнит перечисленные в этом файле команды одну за другой, точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются **сценариями оболочки**. Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут запускаться с параметрами и, кроме того, использовать конструкции `if`, `for`, `while` и `case`. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки. Существует альтернативная оболочка Berkley C, разработанная таким образом, чтобы сценарии оболочки (и команды языка вообще) выглядели во многих аспектах подобно программам на C. Поскольку оболочка представляет собой всего лишь еще одну пользовательскую программу, было написано много различных ее версий.

Утилиты UNIX

Пользовательский интерфейс UNIX состоит не только из оболочки, но также из большого числа стандартных обслуживающих программ, называемых также утилитами. Грубо говоря, эти программы можно разделить на шесть следующих категорий:

1. Команды управления файлами и каталогами.
2. Фильтры.
3. Средства разработки программ, такие как текстовые редакторы и компиляторы.
4. Текстовые процессоры.
5. Системное администрирование.
6. Разное.

Стандарт POSIX 1003.2 определяет синтаксис и семантику менее 100 из этих программ, в основном относящихся к первым трем категориям. Идея стандартизации данных программ заключается в том, чтобы можно было писать сценарии оболочки, которые работали бы на всех системах UNIX. Помимо этих стандартных утилит, разумеется, существует еще масса прикладных программ, таких как web-браузеры, программы просмотра изображений и т. д.

Рассмотрим несколько примеров этих утилит, начиная с программ для управления файлами и каталогами. Команда

```
cp a b
```

копирует файл *a* в *b*, не изменяя исходный файл. Команда

```
mv a b
```

напротив, копирует файл *a* в *b*, но удаляет исходный файл. В результате она не копирует файл, а перемещает его. Несколько файлов можно объединить в один при помощи команды *cat*, считывающей все входные файлы и копирующей их один за

другим в стандартный выходной поток. Удалить файлы можно командой *rm*. Команда *chmod* позволяет владельцу изменить права доступа к файлу. Каталоги можно создать командой *mkdir* и удалить командой *rmdir*. Список файлов можно вывести на экран, принтер или в файл командой *ls*. У этой команды множество флагов (ключей), управляющих видом формируемого ею листинга. При помощи одних флагов можно задать, насколько подробно будет отображаться каждый файл (размер, владелец, группа, дата создания), другими флагами задается порядок, в котором перечисляются файлы (по алфавиту, по времени последнего изменения, в обратном порядке), третья группа флагов позволяет задать расположение списка файлов на экране и т. д.

Мы уже рассматривали несколько примеров фильтров: команда *grep* извлекает из стандартного входного потока или из одного или нескольких файлов строки, содержащие определенную последовательность символов. Команда *sort* сортирует входной поток и выводит отсортированные данные в стандартный выходной поток. Команда *head* пропускает сквозь себя первые несколько строк. Команда *tail*, напротив, выдает на выходе указанное количество последних строк с входа. Кроме того, стандартом 1003.2 определены такие фильтры, как *cut* и *paste*, которые позволяют вырезать и вставлять в файлы куски текста. Команда *od* конвертирует (обычно двоичный) вход в ASCII-строку в восьмеричный, десятичный или шестнадцатеричный формат. Команда *tr* преобразует символы из верхнего регистра в нижний, и наоборот, а команда *pr* форматирует выход для вывода на принтер, позволяя добавлять заголовки, номера страниц и т. д.

Компиляторы и программные средства включают в себя компилятор с языка С *cc* и программу *ar*, собирающую библиотечные процедуры в архивные файлы.

Еще одной важной инструментальной программой является команда *make*, используемая для сборки больших программ, исходный текст которых состоит из нескольких файлов. Как правило, некоторые из этих файлов представляют собой **заголовочные файлы**, содержащие определения типов, переменных, макросов и т. д. Исходные файлы обычно ссылаются на эти файлы с помощью специальной директивы компилятора *include*. Таким образом, два и более исходных файла могут совместно использовать одни и те же определения. Однако если файл заголовков изменен, необходимо найти все исходные файлы, зависящие от него, и перекомпилировать их. Задача команды *make* заключается в том, чтобы отслеживать, какой файл от какого зависит, и автоматически запускать компилятор для тех файлов, которые требуется перекомпилировать. Почти все программы в системе UNIX, кроме самых маленьких, компилируются с помощью команды *make*.

Все упоминавшиеся выше команды перечислены еще раз в табл. 10.1 вместе с кратким описанием. Во всех версиях операционной системы UNIX есть эти программы, а также многие другие.

Таблица 10.1. Некоторые утилиты UNIX, требуемые стандартом POSIX

Программа	Функция
cat	Конкатенация нескольких файлов в стандартный выходной поток
chmod	Изменение режима защиты файла
cp	Копирование файлов
cut	Вырезание колонок текста из файла

продолжение

Таблица 10.1 (*продолжение*)

Программа	Функция
grep	Поиск определенной последовательности символов в файле
head	Извлечение из файла первых строк
ls	Распечатка каталога
make	Компиляция файлов для создания двоичного файла
mkdir	Создание каталога
od	Шестнадцатеричный дамп файла
paste	Вставка колонок текста в файл
pr	Форматирования файла для печати
rm	Удаление файлов
rmdir	Удаление каталога
sort	Сортировка строк файла по алфавиту
tail	Извлечение из файла последних строк
tr	Преобразование символов из одного набора в другой

Структура ядра

На рис. 10.1 была показана общая структура системы UNIX. Давайте теперь поподробнее рассмотрим ядро системы. Обзор структуры ядра системы UNIX представляет собой довольно непростое дело, так как существует множество различных версий этой системы. Однако, хотя диаграмма на рис. 10.2 описывает UNIX 4.4BSD, она также применима ко многим другим версиям, возможно, с небольшими изменениями в тех или иных местах.

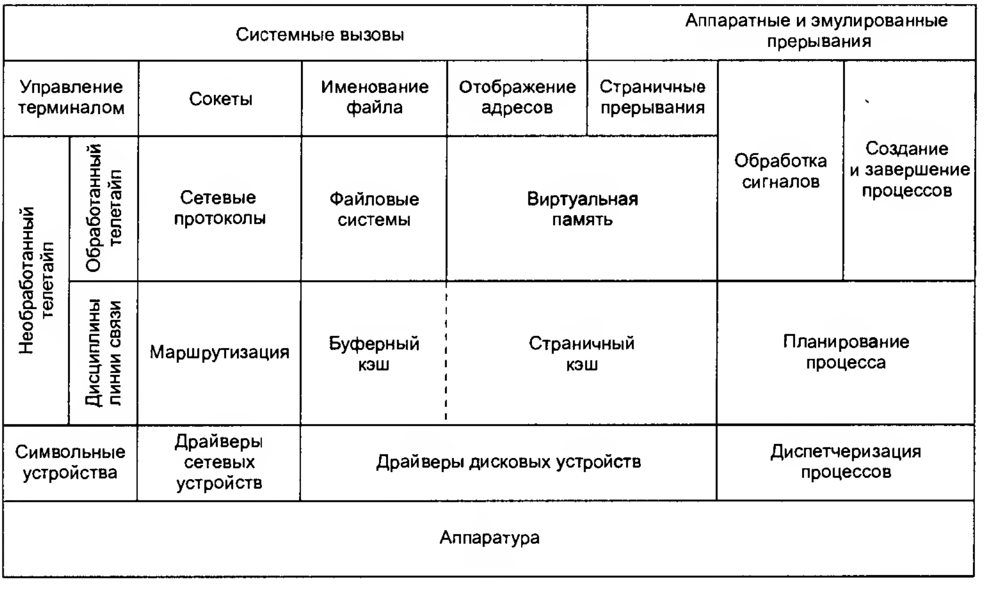


Рис. 10.2. Структура ядра операционной системы UNIX 4.4BSD

Нижний уровень ядра состоит из драйверов устройств и процедуры диспетчеризации процессов. Все драйверы системы UNIX делятся на два класса: драйверы символьных устройств и драйверы блочных устройств. Основное различие между этими двумя классами устройств заключается в том, что на блочных устройствах разрешается операция поиска, а на символьных нет. Технически сетевые устройства представляют собой символьные устройства, но они обрабатываются по-иному, поэтому их, вероятно, правильнее выделить в отдельных класс, как это и было сделано на схеме. Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа останавливает выполнение работающего процесса, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер. Кроме того, диспетчеризация процессов производится также, когда ядро завершает свою работу и пора снова запустить процесс пользователя. Программа диспетчеризации процессов написана на ассемблере и представляет собой отдельную от процедуры планирования программу.

В более высоких уровнях программы отличаются в каждом из четырех «столбцов» диаграммы. Слева располагаются символьные устройства. Они могут использоваться двумя способами. Некоторым программам, таким как текстовые редакторы *vi* и *emacs*, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит ввод-вывод с необработанного терминала (телетайпа). Другое программное обеспечение, например оболочка (*sh*), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша ENTER. Такое программное обеспечение пользуется вводом с терминала в обработанном виде и дисциплинами линии связи.

Сетевое программное обеспечение часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, гарантируя, что правильный пакет направляется правильному устройству или блоку управления протоколами. Большинство систем UNIX содержат в своем ядре полноценный маршрутизатор Интернета, и хотя его производительность ниже, чем у аппаратного маршрутизатора, эта программа появилась раньше современных аппаратных маршрутизаторов. Над уровнем маршрутизации располагается стек протоколов, обязательно включая протоколы IP и TCP, но также иногда и некоторые дополнительные протоколы. Над сетевыми протоколами располагается интерфейс сокетов, позволяющий программам создавать сокеты для отдельных сетей и протоколов. Для использования сокетов пользовательские программы получают дескрипторы файлов.

Над дисковыми драйверами располагаются буферный кэш и страничный кэш файловой системы. В ранних системах UNIX буферный кэш представлял собой фиксированную область памяти, а остальная память использовалась для страниц пользователя. Во многих современных системах UNIX этой фиксированной границы уже не существует, и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости от того, что требуется в данный момент.

Над буферным кэшем располагаются файловые системы. Большинством систем UNIX поддерживаются несколько файловых систем, включая быструю файловую систему Беркли, журнальную файловую систему, а также различные виды файловых систем System V. Все эти файловые системы совместно используют общий буферный кэш. Выше файловых систем помещается именование файлов,

управление каталогами, управление жесткими и символьными связями, а также другие свойства файловой системы, одинаковые для всех файловых систем.

Над страничным кэшем располагается система виртуальной памяти. В нем вся логика работы со страницами, например алгоритм замещения страниц. Поверх него находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. Эта программа решает, что нужно делать при возникновении страничного прерывания. Сначала она проверяет допустимость обращения к памяти и, если все в порядке, определяет местонахождение требуемой страницы и то, как она может быть получена.

Последний столбец имеет отношение к управлению процессами. Над диспетчером располагается планировщик процессов, выбирающий процесс, который должен быть запущен следующим. Если потоками управляет ядро, то управление потоками также помещается здесь, хотя в некоторых системах UNIX управление потоками вынесено в пространство пользователя. Над планировщиком расположена программа для обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.

Верхний уровень представляет собой интерфейс системы. Слева располагается интерфейс системных вызовов. Все системные вызовы поступают сюда и направляются одному из модулей низших уровней в зависимости от природы системного вызова. Правая часть верхнего уровня представляет собой вход для аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.

Процессы в системе UNIX

В предыдущих разделах мы начали наш обзор системы UNIX с точки зрения пользователя, сидящего за терминалом и вводящего команды с клавиатуры. Были приведены примеры часто использующихся команд оболочки и утилит. Этот краткий обзор был завершен рассмотрением структуры системы. Теперь настало время углубиться в ядро и более пристально рассмотреть основные концепции, поддерживаемые системой UNIX, а именно процессы, память, файловую систему и ввод-вывод. Эти сведения важны, так как системные вызовы — интерфейс самой операционной системы — управляют ими. Например, существуют системные вызовы для создания процессов, доступа к памяти, открытия файлов и ввода-вывода.

К сожалению, существует очень много версий системы UNIX и между ними имеются определенные различия. В данной главе основное внимание будет уделено общим чертам всех версий, а не особенностям какой-либо одной версии. Таким образом, в определенных разделах (особенно в тех, где будет рассматриваться вопрос реализации), может оказаться, что описание не соответствует в равной мере всем версиям.

Основные понятия

Единственными активными сущностями в системе UNIX являются процессы. Процессы UNIX очень похожи на классические последовательные процессы, которые мы изучали в главе 2. Каждый процесс запускает одну программу и изначально

получает один поток управления. Другими словами, у процесса есть один счетчик команд, указывающий на следующую исполняемую команду процессора. Большинство версий UNIX позволяют процессу после того, как он запущен, создавать дополнительные потоки.

UNIX представляет собой многозадачную систему, так что несколько независимых процессов могут работать одновременно. У каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Действительно, на большинстве однопользовательских рабочих станций, даже когда пользователь куда-либо отлучается, работают десятки фоновых процессов, называемых **демонами**. Они запускаются автоматически при загрузке системы.

Типичным демоном является *cron daemon*. Он просыпается раз в минуту, проверяя, не нужно ли чего сделать. Если у него есть работа, он ее выполняет и отправляется спать дальше.

Этот демон позволяет планировать в системе UNIX активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться к зубному врачу в 3 часа дня в следующий вторник. Он может создать запись в базе данных демона *cron*, чтобы тот бибикнул ему, скажем, в 2:30. Когда наступает назначенный день, *cron daemon* видит, что у него есть работа, и запускает в назначенное время пищащую программу в виде нового процесса.

Демон *cron* также используется для периодического запуска задач, например ежедневной архивации диска в 4 часа ночи или напоминания забывчивым пользователям каждый год 31 октября купить новые страшненькие товары для веселого празднования Хэллоуина. Другие демоны управляют входящей и исходящей электронной почтой, очередями на принтер, проверяют, достаточно ли еще осталось свободных страниц памяти и т. д. Демоны реализуются в системе UNIX довольно просто, так как каждый из них представляет собой отдельный процесс, независимый ото всех остальных процессов.

Процессы создаются в операционной системе UNIX чрезвычайно несложно. Системный вызов fork создает точную копию исходного процесса, называемого **родительским процессом**. Новый процесс называется **дочерним процессом**. У родительского и у дочернего процессов есть свои собственные образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, изменения остаются невидимыми для дочернего процесса, и наоборот.

Открытые файлы совместно используются родительским и дочерним процессами. Это значит, что если какой-либо файл был открыт до выполнения системного вызова fork, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом, будут видимы каждому процессу. Такое поведение является единственно разумным, так как эти изменения будут также видны любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского процесса и у дочернего идентично, приводит к небольшому затруднению: Как процессам узнать, который из них должен исполнять родительскую программу, а который дочернюю? Эта проблема решается просто: системный вызов fork возвращает дочернему процессу число 0, а родительскому — отличный от нуля **PID** (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса могут проверить возвращаемое значение и действовать соответственно, как показано в листинге 10.1.

Листинг 10.1. Создание процесса в системе UNIX

```
pid = fork( );          /* если fork завершился успешно, pid > 0 в родительском процессе */
if (pid < 0) {
      handle_error();   /* fork потерпел неудачу (например, память или какая-либо таблица
                           переполнена) */
} else if (pid > 0) {
                        /* здесь располагается родительская программа. */
} else {
                        /* здесь располагается дочерняя программа. */
}
```

Процессы распознаются по своим PID-идентификаторам. Как уже говорилось выше, при создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, он может воспользоваться системным вызовом getpid. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его PID также выдается его родителю. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, исходный процесс может создать целое дерево детей, внуков, правнуков и т. д.

В системе UNIX процессы могут общаться друг с другом с помощью разновидности обмена сообщениями. Можно создать канал между двумя процессами, в который один процесс может писать поток байтов, а другой процесс может его читать. Эти каналы иногда называют **трубами**. Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

```
sort <f | head
```

она создает два процесса, *sort* и *head*, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы *sort* соединяется со стандартным потоком ввода программы *head*. При этом все данные, формируемые программой *sort*, попадают напрямую программе *head*, для чего не требуется временного файла. Если канал переполняется, система приостанавливает работу программы *sort*, пока программа *head* не удалит из него хоть сколько-нибудь данных.

Процессы также могут общаться другим способом: при помощи программных прерываний. Один процесс может послать другому так называемый **сигнал**. Процессы могут сообщить системе, какие действия следует предпринимать, когда придет сигнал. У процесса есть выбор: проигнорировать сигнал, перехватить его или позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуры обработки сигналов. Когда сигнал прибывает, управление внезапно передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова возвращается в то место процесса, в котором оно находилось, когда пришел сигнал. Обработка сигналов аналогична обработке аппаратных прерываний ввода-вывода. Процесс может посылать сигналы только членам его **группы процессов**, состоящей из его прямого родителя, всех прародителей,

братьев и сестер, а также детей (внуков и правнуков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит число на 0, он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой). Сигналы, требуемые стандартом POSIX, перечислены в табл. 10.2. В большинстве систем UNIX также имеются дополнительные сигналы, но программы, использующие их, могут оказаться непереносимыми на другие версии UNIX.

Таблица 10.2. Сигналы, требуемые стандартом POSIX

Сигнал	Причина
SIGABRT	Посылается, чтобы прервать процесс и создать дамп памяти
SIGALRM	Истекло время будильника
SIGFPE	Произошла ошибка при выполнении операции с плавающей точкой (например, деление на 0)
SIGHUP	Модем повесил трубку на телефонной линии, использовавшейся процессом
SIGILL	Пользователь нажал клавишу DEL, чтобы прервать процесс
SIGQUIT	Пользователь нажал клавишу, требуя прекращения работы процесса с созданием дампа памяти
SIGKILL	Посылается, чтобы уничтожить процесс (не может игнорироваться или перехватываться)
SIGPIPE	Процесс пишет в канал, из которого никто не читает
SIGSEGV	Процесс обратился к неверному адресу памяти
SIGTERM	Вежливая просьба процессу завершить свою работу
SIGUSR1	Может быть определено приложением
SIGUSR2	Может быть определено приложением

Системные вызовы управления процессами в UNIX

Рассмотрим теперь системные вызовы UNIX, предназначенные для управления процессами. Основные системные вызовы перечислены в табл. 10.3. (В случае ошибки возвращаемое значение *s* равно –1, *pid* означает PID процесса, а *residual* — оставшееся время до предыдущего сигнала будильника.) Обсуждение системных вызовов проще всего начать с системного вызова fork. Этот системный вызов представляет собой единственный способ создания новых процессов в системах UNIX. Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и все остальное. После выполнения системного вызова fork исходный процесс и его копия (родительский процесс и дочерний) идут каждый своим путем. Сразу после выполнения системного вызова fork значение всех соответствующих переменных в обоих процессах одинаково, но затем изменения переменных в одном процессе не влияет на переменные другого процесса. Системный вызов fork возвращает значение, равное нулю для дочернего процесса и идентификатору (PID) дочернего процесса для родительского. Таким образом, два процесса могут определить, кто из них родитель, а кто дочерний процесс.

Таблица 10.3. Некоторые системные вызовы, относящиеся к процессам

Системный вызов	Описание
pid=fork( )	Создать дочерний процесс, идентичный родительскому
pid=waitpid(pid, &statloc, opts)	Ждать завершения дочернего процесса
s=execve(name, argv, envp)	Заменить образ памяти процесса
exit(status)	Завершить выполнение процесса и вернуть статус
s=sigaction(sig, &act, &oldact)	Определить действие, выполняемое при приходе сигнала
s=sigreturn(&context)	Вернуть управление после обработки сигнала
s=sigprocmask(how, &set, &old)	Исследовать или изменить маску сигнала
s=sigpending(set)	Получить или установить блокированные сигналы
s=sigsuspend(sigmask)	Заменить маску сигнала и приостановить процесс
s=kill(pid, sig)	Послать сигнал процессу
residual=alarm(seconds)	Установить будильник
s=pause( )	Приостановить выполнение процесса до следующего сигнала

В большинстве случаев после системного вызова fork дочернему процессу потребуется выполнить программу, отличающуюся от программы, выполняемой родительским процессом. Рассмотрим работу оболочки. Она считывает команды с терминала, с помощью системного вызова fork выполняет введенную команду, затем ждет окончания работы дочернего процесса, после чего считывает следующую команду. Для ожидания завершения дочернего процесса родительский процесс обращается к системному вызову waitpid. У этого системного вызова три параметра. В первом параметре указывается PID процесса, завершение которого ожидается. Если вместо идентификатора процесса указать число −1, то в этом случае системный вызов ожидает завершения любого дочернего процесса. Второй параметр представляет собой адрес переменной, в которую записывается статус завершения дочернего процесса (нормальное или ненормальное завершение, а также возвращаемое на выходе значение). Третий параметр определяет, будет ли обращающийся к системному вызову waitpid процесс блокирован до завершения дочернего процесса или сразу получит управление после обращения к системному вызову.

В случае оболочки дочерний процесс должен выполнить команду, введенную пользователем. Он выполняет это при помощи системного вызова exec, который заменяет весь образ памяти содержимым файла, указанным в первом параметре системного вызова. Крайне упрощенный вариант оболочки, иллюстрирующей использование системных вызовов fork, waitpid и exec, показан в листинге 10.2.

Листинг 10.2. Сильно упрощенная оболочка

```
while (TRUE) {                                      /* вечный цикл */
      type_prompt( );                               /* вывести приглашение ко вводу */
      read_command(command, params);                /* прочитать с клавиатуры строку */
      pid = fork( );                                /* ответвить дочерний процесс */
      if (pid < 0) {
         printf("Создать процесс невозможно");     /* ошибка */
         continue;                                  /* следующий цикл */
      }
```

```
        if (pid != 0) {
            waitpid (-1, &status, 0);         /* родительский процесс ждет завершения
                                              /* дочернего процесса */
        } else {
        execve(command, params, 0);           /* дочерний процесс выполняет работу */
        }
}
```

В самом общем случае у системного вызова exec три параметра: имя исполняемого файла, указатель на массив аргументов и указатель на массив строк окружения. Различные варианты этой процедуры, включая *execl*, *execv*, *execle* и *execve*, позволяют опускать некоторые параметры или указывать их иными способами. Все эти процедуры обращаются к одному и тому же системному вызову. Хотя сам системный вызов называется exec, библиотечной процедуры с таким именем нет.

Рассмотрим случай выполнения оболочкой команды

```
cp file1 file2
```

используемой для копирования файла *file1* в файл *file2*. После того как оболочка создает дочерний процесс, тот обнаруживает и исполняет файл *cp* и передает ему информацию о копируемых файлах.

Головной модуль файла *cp* (как и многие другие программы) содержит определение функции

```
main(argc, argv, envp)
```

где *argc* — счетчик слов (последовательностей символов, ограниченных пробелами) в командной строке, включая имя программы. Для вышеприведенного примера значение *argc* равно 3.

Второй параметр *argv* представляет собой указатель на массив. *i*-й элемент этого массива является указателем на *i*-е слово командной строки. В нашем примере элемент *argv*[0] указывает на строку «ср». Соответственно, элемент *argv*[1] указывает на строку «file1», а элемент *argv*[2] указывает на строку «file2».

Третий параметр процедуры *main*, *envp*, представляет собой указатель на переменные среды и является массивом, содержащим строки вида *имя = значение*, используемые для передачи программе такой информации, как тип терминала и имя рабочего каталога. В листинге 10.2 дочернему процессу переменные среды не передаются, поэтому третий параметр *envp* в данном случае равен нулю.

Если системный вызов exec показался вам слишком сложным, не отчаивайтесь. Это самый сложный системный вызов. Все остальные значительно проще. В качестве примера простого системного вызова рассмотрим exit, который процессы должны использовать, заканчивая исполнение. У него есть один параметр, статус выхода (от 0 до 255), возвращаемый родительскому процессу в переменной *status* системного вызова waitpid. Младший байт переменной *status* содержит статус завершения, равный 0 при нормальном завершении или коду ошибки при аварийном завершении. Например, если родительский процесс выполняет оператор

```
n = waitpid(-1, &status, 0);
```

он будет приостановлен до тех пор, пока не завершится какой-либо дочерний процесс. Если дочерний процесс завершится со, скажем, значением статуса, равным 4,

в качестве параметра библиотечной процедуры *exit*, то родительский процесс получит PID дочернего процесса и значение статуса, равное 0x0400 (0x означает в программах на языке C шестнадцатеричное число). Младший байт переменной *status* относится к сигналам, старший байт представляет собой значение, задаваемое дочерним процессом в виде параметра при обращении к системному вызову `exit`.

Если процесс уже завершил свою работу, а родительский процесс не ожидает этого события, то дочерний процесс переводится в так называемое **состояние зомби**, то есть приостанавливается. Когда родительский процесс наконец обращается к библиотечной процедуре *waitpid*, дочерний процесс завершается.

Несколько системных вызовов относятся к сигналам, используемым различными способами. Например, если пользователь ненамеренно велит текстовому редактору отобразить содержание очень длинного файла, а затем осознает свою ошибку, то потребуется некий способ прервать работу редактора. Обычно для этого пользователь нажимает специальную клавишу (например, `DEL` или `CTRL+C`), в результате чего редактору посылается сигнал. Редактор перехватывает сигнал и останавливает вывод.

Чтобы заявить о своем желании перехватить тот или иной сигнал, процесс может воспользоваться системным вызовом `sigaction`. Первый параметр этого системного вызова — сигнал, который требуется перехватить (см. табл. 10.2). Второй параметр представляет собой указатель на структуру, в которой хранится указатель на процедуру обработки сигнала вместе с различными битами и флагами. Третий параметр указывает на структуру, в которой система возвращает информацию о текущем обрабатываемом сигнале, на случай, если позднее его нужно будет восстановить.

Обработчик сигнала может выполняться сколь угодно долго. Однако на практике обработка сигналов не занимает много времени. Когда процедура обработки сигнала завершает свою работу, она возвращается к той точке, в которой ее прервали.

Системный вызов `sigaction` может также использоваться для игнорирования сигнала или чтобы восстановить действие по умолчанию, заключающееся в уничтожении процесса.

Нажатие на клавишу `DEL` не является единственным способом послать сигнал. Системный вызов `kill` позволяет процессу послать сигнал любому родственному процессу. Выбор названия для данного системного вызова (kill — убить, уничтожить) не особенно удачен, так как по большей части он используется процессами не для уничтожения других процессов, а, наоборот, в надежде, что этот сигнал будет перехвачен и обработан соответствующим образом.

Во многих приложениях реального времени бывает необходимо прервать процесс через определенный интервал времени, чтобы заставить его сделать что-либо, например переслать повторно возможно потерянный пакет по ненадежной линии связи. Для обработки данной ситуации предоставлен системный вызов `alarm` (будильник). Параметр этого системного вызова задает временной интервал, по истечении которого процессу посылается сигнал SIGALRM. У процесса в каждый

момент времени может быть только один будильник. Например, если процесс обращается к системному вызову alarm с параметром 10 с, а 3 с спустя снова обращается к нему с параметром 20 с, то он получит только один сигнал через 20 с после второго системного вызова. Первый сигнал будет отменен вторым обращением к системному вызову alarm. Если параметр системного вызова alarm равен нулю, то такое обращение отменяет любой сигнал будильника. Если сигнал будильника не перехватывается, то действие по умолчанию заключается в уничтожении процесса. Технически возможно игнорирование данного сигнала, но смысла такое действие не имеет.

Иногда случается так, что процессу нечем заняться, пока не придет сигнал. Рассмотрим, например, обучающую программу, проверяющую скорость чтения и понимание текста. Она отображает на экране некоторый текст, после чего обращается к системному вызову alarm, чтобы система послала ей сигнал через 30 с. Пока студент читает текст, у программы нет дел. Она может сидеть в коротком цикле, ничего не делая, но такая реализация будет напрасно расходовать время центрального процессора, которое может понадобиться фоновому процессу или другому пользователю. Лучшее решение заключается в использовании системного вызова pause, велящего операционной системе UNIX приостановить работу процесса, пока не прибудет следующий сигнал.

Системные вызовы управления потоками

В первой версии системы не было потоков. Это свойство было добавлено много лет спустя. Изначально применялось множество различных пакетов поддержки потоков, однако распространение этих различных пакетов привело к тому, что написать переносимую программу стало очень сложно. В конце концов, системные вызовы, используемые для управления потоков, были стандартизированы в виде части стандарта POSIX (P1003.1c).

В стандарте POSIX не указывается, должны ли потоки реализовываться в пространстве ядра или в пространстве пользователя. Преимущество потоков в пользовательском пространстве состоит в том, что они легко реализуются беэ необходимости изменения ядра, а переключение потоков осуществляется очень эффективно. Недостаток потоков в пространстве пользователя заключается в том, что если один из потоков заблокируется (например, на операции ввода-вывода, семафоре или страничном прерывании), все потоки процесса блокируются. Ядро полагает, что существует только один поток, и не передает управление процессу потока, пока блокировка не снимется. Таким образом, системные вызовы, определенные в стандарте P1003.1c, были тщательно отобраны так, чтобы потоки могли быть реализованы любым способом. До тех пор пока пользовательские программы четко придерживаются семантики стандарта P1003.1c, оба способа реализации должны работать корректно. Наиболее часто применяемые вызовы управления потоками перечислены в табл. 10.4. Когда используется системная реализация потоков, они являются настоящими системными вызовами. При использовании потоков на уровне пользователя они полностью реализуются в динамической библиотеке в пространстве польэователя.

Таблица 10.4. Основные вызовы управления потоками стандарта POSIX

Вызов	Описание
pthread_create	Создать новый поток в адресном пространстве вызывающего процесса
pthread_exit	Завершить вызывающий процесс
pthread_join	Подождать, пока не завершится процесс
pthread_mutex_init	Создать новый мьютекс
pthread_mutex_destroy	Уничтожить мьютекс
pthread_mutex_lock	Заблокировать мьютекс
pthread_mutex_unlock	Разблокировать мьютекс
pthread_cond_init	Создать условную переменную
pthread_cond_destroy	Уничтожить условную переменную
pthread_cond_wait	Ждать условную переменную
pthread_cond_signal	Разблокировать один поток, ждущий условную переменную

Для действительно внимательных читателей отметим, что теперь у нас появляется типографская проблема. Когда управление потоками осуществляется в ядре, тогда такие вызовы, как «pthread_create», являются системными вызовами и, следуя нашим традициям, должны набираться моноширинным шрифтом: `pthread_create`. Однако если это просто библиотечные процедуры в пространстве пользователя, то нам следует использовать для них курсивный шрифт, как здесь: *pthread_create*. Для простоты мы будем везде использовать рубленый шрифт, особенно в следующей главе, в которой никогда точно не ясно, какие из вызовов Win32 API являются настоящими системными вызовами. Могло быть и хуже: в языке Algol 68 Report существовала точка, печать которой не тем шрифтом слегка меняла грамматику языка.

Давайте кратко рассмотрим вызовы управления потоками, показанные в табл. 10.4. Первый вызов `pthread_create` создает новый поток. Обращение к этому системному вызову производится следующим образом:

```
err = pthread_create(&tid, attr, function, arg);
```

Этот вызов создает в текущем процессе новый поток, в котором работает программа *function*, а *arg* передается этой программе в качестве параметра. Идентификатор нового потока хранится в памяти по адресу, на который указывает первый параметр. С помощью параметра *attr* можно задавать для нового потока определенные атрибуты, такие как приоритет планирования. После успешного выполнения данного системного вызова в адресном пространстве пользователя появляется на один поток больше.

Поток, выполнивший свою работу и желающий прекратить свое существование, обращается к системному вызову `pthread_exit`. Поток может подождать, пока не завершится процесс, обратившись к системному вызову `pthread_join`. Если ожидаемый поток уже завершил свою работу, системный вызов `pthread_join` выполняется мгновенно. В противном случае обратившийся к нему поток блокируется.

Синхронизация потоков может осуществляться при помощи **мьютексов**. Как правило, мьютекс охраняет какой-либо ресурс, например буфер, совместно используемый двумя потоками. Чтобы гарантировать, что только один поток в каждый момент времени имеет доступ к общему ресурсу, предполагается, что потоки

блокируют (захватывают) мьютекс перед обращением к ресурсу и разблокируют (отпускают) его, когда ресурс им более не нужен. До тех пор пока потоки соблюдают данный протокол, состояния состязания можно избежать. Мьютексы подобны двоичным семафорам, то есть семафорам, способным принимать только значения 0 и 1. Название мьютекс (mutex) образовано от английских слов mutual exclusion — взаимное исключение.

Мьютексы могут создаваться вызовом pthread_mutex_init и уничтожаться при помощи вызова pthread_mutex_destroy. Мьютекс может находиться в одном из двух состояний: блокированный и разблокированный. Поток может заблокировать мьютекс с помощью вызова pthread_mutex_lock. Если мьютекс уже заблокирован, то поток, обратившийся к этому вызову, блокируется. Когда поток, захвативший мьютекс, выполнил свою работу в критической области, он должен освободить мьютекс, обратившись к вызову pthread_mutex_unlock.

Мьютексы предназначены для кратковременной блокировки, например для защиты совместно используемой переменной. Они не предназначаются для долговременной синхронизации, например для ожидания, когда освободится накопитель на магнитной ленте. Для долговременной синхронизации предоставляются **переменные состояния**. Эти переменные создаются с помощью вызова pthread_cond_init и уничтожаются вызовом pthread_cond_destroy.

Переменные состояния используются следующим образом: один поток ждет, когда переменная примет определенное значение, а другой поток сигнализирует ему изменением этой переменной. Например, обнаружив, что нужный ему накопитель на магнитной ленте занят, поток может обратиться к вызову pthread_cond_wait, задав в качестве параметра адрес переменной, которую все потоки согласились связать с накопителем на магнитной ленте. Когда поток, использующий накопитель на магнитной ленте, наконец, освободит это устройство (возможно, через несколько часов), он обращается к вызову pthread_cond_signal, чтобы изменить переменную состояния и тем самым сообщить ожидающим потокам, что магнитофон свободен. Если ни один поток в этот момент не ждет, когда освободится накопитель на магнитной ленте, этот сигнал просто теряется. Другими словами, переменные состояния не считаются семафорами. С потоками, мьютексами и переменными состояния также определены несколько других операций.

Реализация процессов в UNIX

Процесс в системе UNIX подобен айсбергу: то, что вы видите, представляет собой всего лишь выступающую над водой его часть, но не менее важная часть скрыта под водой. У каждого процесса есть пользовательская часть, в которой работает программа пользователя. Однако когда один из потоков обращается к системному вызову, происходит эмулированное прерывание с переключением в режим ядра. После этого поток начинает работу в контексте ядра, с отличной картой памяти и полным доступом к ресурсам машины. Это все еще тот же самый поток, но теперь обладающий большей мощью, а также со своим стеком ядра и счетчиком команд в режиме ядра. Это важно, так как системный вызов может блокироваться на полпути, например, ожидая завершения дисковой операции. При этом счетчик

команд и регистры будут сохранены таким образом, чтобы позднее поток можно было восстановить в режиме ядра.

Ядро поддерживает две ключевые структуры данных, относящиеся к процессам: **таблицу процессов** и **структуру пользователя**. Таблица процессов является резидентной. В ней содержится информация, необходимая для всех процессов, даже для тех процессов, которых в данный момент нет в памяти. Структура пользователя выгружается на диск, освобождая место в памяти, когда относящийся к ней процесс отсутствует в памяти, чтобы не тратить память на ненужную в данный момент информацию.

Информация в таблице процессов подразделяется на следующие категории:

1. **Параметры планирования.** Приоритеты процессов, процессорное время, потребленное за последний учитываемый период, количество времени, проведенное процессом в режиме ожидания. Вся эта информация используется для выбора процесса, которому будет передано управление следующим.
2. **Образ памяти.** Указатели на сегменты программы, данных и стека, или, если используется страничная организация памяти, указатели на соответствующие им таблицы страниц. Если программный сегмент используется совместно, то программный указатель указывает на общую таблицу программы. Когда процесса нет в памяти, то здесь также содержится информация о том, как найти части процесса на диске.
3. **Сигналы.** Маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.
4. **Разное.** Текущее состояние процесса, события, ожидаемые процессом (если таковые есть), время до истечения интервала будильника, PID процесса, PID родительского процесса, идентификаторы пользователя и группы.

В структуре пользователя содержится информация, которая не требуется, когда процесса физически нет в памяти и он не выполняется. Например, хотя процессу, выгруженному на диск, можно послать сигнал, выгруженный процесс не может прочитать файл. По этой причине информация о сигналах должна храниться в таблице процессов, постоянно находящейся в памяти, даже когда процесс не присутствует в памяти. С другой стороны, сведения об описателях файлов могут храниться в структуре пользователя и загружаться в память вместе с процессом.

Данные, хранящиеся в структуре пользователя, включают в себя следующие пункты:

1. **Машинные регистры.** Когда происходит прерывание с переключением в режим ядра, машинные регистры (включая регистры с плавающей точкой) сохраняются здесь.
2. **Состояние системного вызова.** Информация о текущем системном вызове, включая параметры и результаты.
3. **Таблица дескрипторов файлов.** Когда происходит обращение к системному вызову, работающему с файлом, дескриптор файла используется в качестве индекса в данной таблице, что позволяет найти структуру данных (i-узел), соответствующую данному файлу.

4. **Учетная информация.** Указатель на таблицу, учитывающую процессорное время, использованное процессом в пользовательском и системном режимах. В некоторых системах здесь также ограничивается процессорное время, которое может использовать процесс, максимальный размер стека, количество страниц памяти и т. д.
5. **Стек ядра.** Фиксированный стек для использования процессом в режиме ядра.

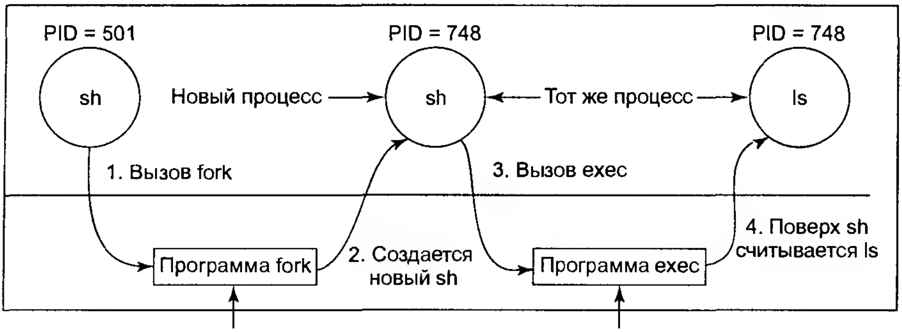
Теперь, зная, что хранится в данных таблицах, легко объяснить, как в системе UNIX создаются процессы. Когда выполняется системный вызов fork, вызывающий процесс обращается в ядро и ищет свободную ячейку в таблице процессов, в которую можно записать данные о дочернем процессе. Если свободная ячейка находится, системный вызов копирует туда информацию из ячейки родительского процесса. Затем он выделяет память для сегментов данных и для стека дочернего процесса, куда копируются соответствующие сегменты родительского процесса. Структура пользователя (которая часто хранится вместе с сегментом стека) копируется вместе со стеком. Программный сегмент может либо копироваться, либо использоваться совместно, если он доступен только для чтения. Начиная с этого момента дочерний процесс может быть запущен.

Когда пользователь вводит с терминала команду, например ls, оболочка создает новый процесс, клонируя свой собственный процесс с помощью системного вызова fork. Новый процесс оболочки затем вызывает системный вызов exec, чтобы считать в свою область памяти содержимое исполняемого файла *ls*. Эти действия показаны на рис. 10.3.

Механизм создания нового процесса довольно прост. Для дочернего процесса создается новая ячейка в таблице процессов, которая заполняется по большей мере из соответствующей ячейки родительского процесса. Дочерний процесс получает PID, затем настраивается его карта памяти. Кроме того, дочернему процессу предоставляется совместный доступ к файлам родительского процесса. Затем настраиваются регистры дочернего процесса, после чего он готов к запуску.

В принципе, следует создать полную копию адресного пространства, так как семантика системного вызова fork говорит, что никакая область памяти не используется совместно родительским и дочерним процессами. Однако копирование памяти является дорогим удовольствием, поэтому все системы UNIX слегка жульничают. Они выделяют дочернему процессу новые таблицы страниц, но эти таблицы указывают на страницы родительского процесса, помеченные как доступные только для чтения. Когда дочерний процесс пытается писать в такую страницу, происходит прерывание. При этом ядро выделяет дочернему процессу новую копию этой страницы, к которой этот процесс получает также и доступ записи. Таким образом, копируются только те страницы, в которые дочерний процесс пишет новые данные. Такой механизм называется **копированием при записи**. При этом сохраняется память, так как страницы с программой не копируются.

После того как дочерний процесс начинает работу, его программа (копия оболочки) выполняет системный вызов exec, задавая имя команды в качестве параметра. При этом ядро находит и проверяет исполняемый файл, копирует в ядро аргументы и строки окружения, а также освобождает старое адресное пространство и его таблицы страниц.



Выделить память для элемента таблицы дочернего процесса
Заполнить элемент таблицы дочернего процесса из элемента родительского процесса
Выделить память для стека и области пользователя дочернего процесса
Заполнить область пользователя дочернего процесса из соответствующей области
Выделить PID для дочернего процесса
Настроить дочерний процесс на использование программы родительского процесса
Копировать таблицы страниц для данных и стека
Настроить совместное использование открытых файлов
Копировать регистры родительского процесса в дочерний

Найти исполняемый файл
Проверить разрешение на выполнение
Прочитать и проверить заголовок
Копировать аргументы, среду в ядро
Освободить новое адресное пространство
Копировать аргументы, среду в стек
Сбросить сигналы
Инициализировать регистры

Рис. 10.3. Этапы выполнения команды *ls*, введенной в оболочке

После этого следует создать и заполнить новое адресное пространство. Если системой поддерживается отображение файлов на адресное пространство памяти, как, например, в System V, BSD и в большинстве других версий UNIX, то таблицы страниц настраиваются следующим образом: в них указывается, что страниц в памяти нет, кроме, возможно, одной страницы со стеком, а содержимое адресного пространства может подгружаться из исполняемого файла на диске. Когда новый процесс начинает работу, он немедленно вызывает страничное прерывание, в результате которого первая страница программы подгружается с диска. Таким образом, ничего не нужно загружать заранее, что позволяет быстро запускать программы, а в память загружать только те страницы, которые действительно нужны программам. Наконец, в стек копируются аргументы и строки окружения, сигналы сбрасываются, а все регистры устанавливаются на ноль. С этого момента новая команда начинает исполнение.

Потоки в UNIX

Реализация потоков зависит от того, поддерживаются они ядром или нет. Если потоки ядром не поддерживаются, как, например, в 4BSD, реализация потоков целиком осуществляется в библиотеке, загружающейся в пространстве пользова-

теля. Если ядро поддерживает потоки, как в системах System V и Solaris, то у ядра есть чем заняться. Потоки обсуждались в общих чертах в главе 2. Здесь мы просто сделаем несколько замечаний по поводу реализации потоков в ядре в системе UNIX.

Основная проблема реализации потоков заключается в поддержке корректной традиционной семантики UNIX. Рассмотрим сначала системный вызов fork. Предположим, что процесс с несколькими потоками (реализуемыми в ядре) выполняет системный вызов fork. Следует ли при этом в новом процессе создать все потоки оригинального процесса? Предположим, что мы ответили на этот вопрос утвердительно. Допустим также, что один из потоков был блокирован, ожидая ввода с клавиатуры. Должна ли в этом случае копия этого потока также быть блокирована ожиданием ввода с клавиатуры? Если да, то какому потоку достанется следующая, набранная на клавиатуре строка? Если нет, то что должен делать этот поток в новом процессе? Проблема касается и других аспектов. В однопоточном процессе такой проблемы не возникает, так как единственный поток не может быть блокирован при обращении к системному вызову fork. Теперь рассмотрим случай, при котором в дочернем процессе другие потоки не создаются. Предположим, что один из некопируемых потоков удерживает мьютекс, который пытается получить единственный созданный новый поток после выполнения системного вызова fork. В этом случае мьютекс никогда не будет освобожден, и новый поток повиснет навсегда. Существует еще множество подобных проблем. И простого решения у этих проблем нет.

Файловый ввод-вывод представляет собой еще одну проблемную область. Предположим, что один поток блокирован при чтении из файла, а другой поток закрывает файл и обращается к системному вызову lseek, чтобы изменить текущий указатель файла. Что произойдет в результате этих действий? Кто знает?

Обработка сигналов тоже представляет собой сложный вопрос. Должны ли сигналы направляться определенному потоку или всему процессу в целом? Вероятно, сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой) должен перехватываться тем потоком, который его вызвал. Что следует делать, если он его не перехватывает? Следует ли убить этот поток? Следует ли убить все потоки процесса? Рассмотрим теперь сигнал SIGINT, посылаемый пользователем, когда он нажимает на определенную клавишу. Какой поток должен перехватывать этот сигнал? Должны ли у всех потоков быть общие маски сигналов? Любые попытки вытянуть нос в одном месте приводят к тому, что в каком-либо другом месте увязает хвост. Корректная реализация семантики потоков (не говоря уже о программе) представляет собой нетривиальную задачу.

Потоки в системе Linux

Операционная система Linux поддерживает потоки в ядре довольно интересным способом, с которым следует познакомиться. В основе реализации системы Linux лежат идеи из системы 4.4BSD, но в 4.4BSD потоки на уровне ядра реализованы не были, так как у университета Калифорнии в Беркли кончились деньги прежде, чем библиотеки языка С могли быть переписаны так, чтобы решить все описанные выше проблемы.

Сердцем реализации потоков в системе Linux является новый системный вызов clone, отсутствующий во всех остальных версиях системы UNIX. Формат обращения к нему выглядит следующим образом:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Системный вызов clone создает новый поток либо в текущем процессе, либо в новом процессе, в зависимости от флага *sharing_flags*. Если новый поток находится в текущем процессе, он совместно использует с остальными потоками адресное пространство и любое изменение каждого байта в адресном пространстве любым потоком тут же становится видимым всем остальным потокам процесса. С другой стороны, если адресное пространство не используется совместно, тогда новый поток получает точную копию адресного пространства, но последующие изменения в памяти уже не видны остальным потокам. Таким образом, здесь используется та же семантика, что и у системного вызова fork.

В обоих случаях новый поток начинает выполнение функции *function* с аргументом *arg* в качестве параметра. Также в обоих случаях новый поток получает свой собственный стек, при этом указатель стека инициализируется параметром *stack_ptr*.

Параметр *sharing_flags* представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного использования, нежели используется в традиционных системах UNIX. У этого флага определены пять битов, перечисленные в табл. 10.5. Каждый бит управляет одним из аспектов совместного использования, и каждый из битов может быть установлен независимо от остальных битов. Бит *CLONE_VM* определяет, будет ли виртуальная память (то есть адресное пространство) использоваться совместно со старыми потоками или будет копироваться. Если этот бит установлен, новый поток просто помещается вместе со старыми потоками, так что системный вызов clone создает новый поток в существующем процессе. Если бит сброшен, новый поток получает свое собственное адресное пространство. Это означает, что результат команды процессора STORE не виден остальным потокам. Такое поведение подобно поведению системного вызова fork. Создание нового адресного пространства равнозначно определению нового процесса.

Таблица 10.5. Биты массива sharing_flags

Флаг	Значение в 1	Значение в 0
CLONE_VM	Создать новый поток	Создать новый процесс
CLONE_FS	Общие рабочий каталог, каталог root и umask	Не использовать их совместно
CLONE_FILES	Общие дескрипторы файлов	Копировать дескрипторы файлов
CLONE_SIGHAND	Общая таблица обработчика сигналов	Копировать таблицу
CLONE_PID	Новый поток получает старый PID	Новый поток получает новый PID

Бит *CLONE_FS* управляет совместным использованием рабочего каталога и каталога root, а также флага umask. Даже если у нового потока свое собственное адресное пространство, при установленном бите *CLONE_FS* старый и новый потоки будут совместно использовать рабочие каталоги. Это означает, что обращение к системному вызову chdir одним из потоков изменит рабочий каталог другого

потока, несмотря на то что у другого потока есть свое собственное адресное пространство. В системе UNIX обращение к системному вызову chdir потоком всегда изменяет рабочий каталог всех остальных потоков этого процесса, но никогда не меняет рабочих каталогов других процессов. Таким образом, этот бит обеспечивает разновидность совместного использования, недоступную в UNIX.

Бит *CLONE_FILES* аналогичен биту *CLONE_FS*. Если он установлен, то новый поток пользуется теми же дескрипторами файлов, что и старые потоки. Таким образом, обращение к системному вызову lseek одним потоком становится видимым для других потоков, что также обычно справедливо для потоков одного процесса, но не для потоков различных процессов. Аналогично бит *CLONE_SIGHAND* разрешает или запрещает совместное использование таблицы обработчиков сигналов старым и новым потоками. Если таблица общая даже у потоков в различных адресных пространствах, тогда изменение обработчика в одном потоке повлияет и на другой поток. Наконец, бит *CLONE_PID* указывает, получит ли новый поток свой собственный PID или будет использовать PID своего родительского потока. Это свойство нужно при загрузке системы. Процессам пользователя не разрешается использовать этот бит.

Такая детализация в вопросе совместного использования стала возможна благодаря тому, что в системе Linux для различных вопросов, перечисленных в начале раздела «Реализация процессов в UNIX» данной главы (параметры планирования, образ памяти и т. д.), используются различные структуры данных. Таблица процессов и структура пользователя просто содержат указатели на эти структуры данных, поэтому легко создать новый элемент таблицы для каждого клонированного потока и сделать так, чтобы он указывал либо на старую структуру, управляющую планированием потоков, памятью или еще чем-либо, либо на копию такой структуры. Сам факт наличия такой высокой степени детализации совместного использования еще не означает, что она полезна, особенно учитывая, что в системе UNIX это не поддерживается. Если какая-либо программа в системе Linux пользуется этим преимуществом, это означает, что она не может без переделок работать в системе UNIX.

Планирование в системе UNIX

Давайте теперь изучим алгоритм планирования системы UNIX. Поскольку UNIX всегда была многозадачной системой, ее алгоритм планирования с самого начала развития системы разрабатывался так, чтобы обеспечить хорошую реакцию в интерактивных процессах. У этого алгоритма два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти и готовых к работе. Высокоуровневый алгоритм перемещает процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными.

У каждой версии UNIX свой слегка отличающийся низкоуровневый алгоритм планирования, но у большинства этих алгоритмов есть много общих черт, которые мы здесь и опишем. В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя (верхняя часть айсберга), имеют положительные значения приоритетов. У процессов, выполняющихся в режиме ядра (обращающихся к системным вызовам), значения приоритетов

отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные — наоборот, минимальными, как показано на рис. 10.4. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

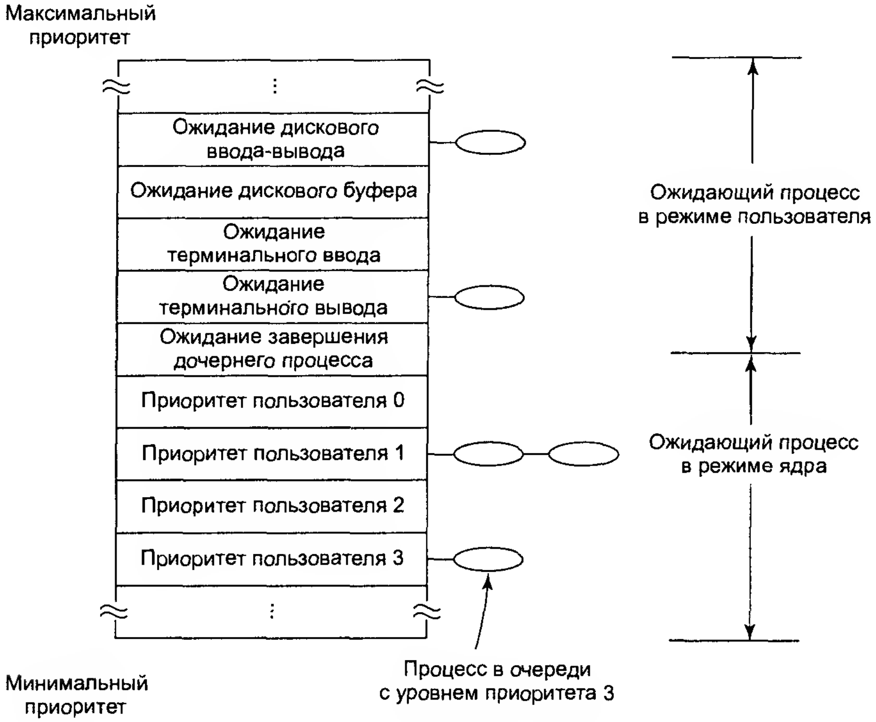


Рис. 10.4. Планировщик системы UNIX основан на структуре многоуровневой очереди

Когда запускается (низкоуровневый) планировщик, он ищет очередь, начиная с самого высокого приоритета (то есть с наименьшего отрицательного значения), пока не находит очередь, в которой есть хотя бы один процесс. После этого в этой очереди выбирается и запускает первый процесс. Ему разрешается работать в течение некоего максимального кванта времени, как правило, 100 мс, или пока он не заблокируется. Если процесс использует весь свой квант времени, он помещается обратно, в конец очереди, а алгоритм планирования запускается снова. Таким образом, процессы, входящие в одну группу приоритетов, совместно используют центральный процессор в порядке циклической очереди.

Раз в секунду приоритет каждого процесса пересчитывается по формуле, состоящей из трех компонентов:

$$priority = CPU_usage + nice + base.$$

На основе сосчитанного нового приоритета каждый процесс прикрепляется к соответствующей очереди на рис. 10.4. Для получения номера очереди приоритет, как правило, делится на некую константу. Изучим вкратце каждый из трех компонентов этой формулы приоритета.

Параметр *CPU_usage* (использование центрального процессора) представляет собой среднее значение тиков таймера в секунду, которые процесс работал в течение последних нескольких секунд. При каждом тике (прерывании) таймера счетчик использования центрального процессора в таблице процессов увеличивается на единицу. Этот счетчик в конце концов добавляется к приоритету процесса, увеличивая тем самым числовое значение его приоритета (что соответствует более низкому приоритету), в результате чего процесс попадает в менее приоритетную очередь.

Однако операционная система UNIX не наказывает процесс за использование центрального процессора навечно, и величина *CPU_usage* со временем уменьшается. В различных версиях UNIX это уменьшение выполняется по-разному. Один из способов состоит в том, что к *CPU_usage* прибавляется полученное число тиков ΔT, после чего сумма делится на два. Такой алгоритм учитывает самое последнее значение ΔT с весовым коэффициентом 1/2, предшествующее ему — с весовым коэффициентом 1/4 и т. д. Алгоритм взвешивания очень быстр, так как состоит из всего одной операции сложения и одного сдвига, но также применяются и другие схемы взвешивания.

С каждым процессом связано значение *nice*. Его значение по умолчанию равно 0, но допустимый диапазон значений, как правило, составляет от –20 до +20. Процесс может установить значение *nice* в диапазоне от 0 до 20 с помощью системного вызова `nice`[1]. Пользователь, вычисляющий в фоновом режиме число π с миллиардом знаков после точки, может обратиться к этому системному вызову, чтобы быть вежливым по отношению к другим пользователям. Только системный администратор может запросить обслуживание с *более высоким* приоритетом (то есть значения *nice* от –20 до –1). О причине введения такого правила читателю предлагается догадаться самому.

Когда процесс эмулирует прерывание для выполнения системного вызова в ядре, процесс, вероятно, должен быть блокирован, пока системный вызов не будет выполнен и не вернется в режим пользователя. Например, процесс может обратиться к системному вызову `waitpid`, ожидая, пока один из его дочерних процессов не закончит работу. Он может также ожидать ввода с терминала или завершения дисковой операции ввода-вывода и т. д. Когда процесс блокируется, он удаляется из структуры очереди, пока этот процесс снова не будет готов работать.

Однако когда происходит событие, которого ждал процесс, он снова помещается в очередь с отрицательным значением. Выбор очереди определяется событием, которого ждал процесс. На рис. 10.4 дисковый ввод-вывод показан как событие с наивысшим приоритетом, так что процесс, только что прочитавший или записавший блок диска, вероятно, получит центральный процессор в течение 100 мс. Отрицательные значения приоритета для дискового ввода-вывода, терминального ввода-вывода и т. д. жестко прошиты в операционной системе и могут быть изменены только путем перекомпиляции самой системы. Эти (отрицательные) значения представлены в приведенной выше формуле слагаемым *base* (база), и их величина достаточно отличается от нуля, чтобы перезапущенный процесс наверняка попадал в другую очередь.

---

[1] В английском языке это слово имеет много значений. В данном случае его можно перевести, как «тактичный», «внимательный», «любезный». — *Примеч. перев.*

В основе этой схемы лежит идея как можно более быстрого удаления процессов из ядра. Если процесс пытается читать дисковый файл, необходимость ждать целую секунду между обращениями к системным вызовам read замедлит его работу во много раз. Значительно лучше позволить ему немедленно продолжить работу сразу после выполнения запроса, так чтобы он мог быстро обратиться к следующему системному вызову. Если процесс был заблокирован ожиданием ввода с терминала, то, очевидно, это интерактивный процесс, и ему должен быть предоставлен наивысший приоритет, как только он перейдет в состояние готовности, чтобы гарантировать хорошее качество обслуживания интерактивных процессов. Таким образом, процессы, ограниченные производительностью процессора (то есть находящиеся в положительных очередях), в основном обслуживаются после того, как будут обслужены все процессы, ограниченные вводом-выводом (когда все эти процессы окажутся заблокированы в ожидании ввода-вывода).

Планирование в системе Linux

Планирование представляет собой одну из немногих областей, в которых операционная система Linux использует алгоритм, отличный от применяющегося в UNIX. Мы только что рассмотрели алгоритм планирования системы UNIX, теперь познакомимся с алгоритмом планирования системы Linux. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах. В операционной системе Linux алгоритмом планирования различаются три класса потоков:

1. Потоки реального времени, обслуживаемые по алгоритму FIFO (First in First Out — первым прибыл — первым обслужен).
2. Потоки реального времени, обслуживаемые в порядке циклической очереди.
3. Потоки разделения времени

Потоки реального времени, обслуживаемые по алгоритму FIFO, имеют наивысшие приоритеты и не могут прерываться другими потоками, за исключением такого же потока реального времени FIFO, перешедшего в состояние готовности. Потоки реального времени, обслуживаемые в порядке циклической очереди, представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они могут прерываться таймером. Находящиеся в состоянии готовности потоки реального времени, обслуживаемые в порядке циклической очереди, выполняются в течение определенного кванта времени, после чего поток помещается в конец своей очереди. Ни один из этих классов на самом деле не является классом реального времени. Здесь нельзя задать предельный срок выполнения задания и предоставить гарантий его выполнения. Эти классы просто имеют более высокий приоритет, чем у потоков стандартного класса разделения времени. Причина, по которой в системе Linux эти классы называются классами реального времени, в том, что операционная система Linux совместима со стандартом P1003.4 (расширение «реального времени» для UNIX), в котором они носят эти имена.

У каждого потока есть приоритет планирования. Значение по умолчанию равно 20, но оно может быть изменено при помощи системного вызова `nice(value)`, вычитающего значение *value* из 20. Поскольку *value* должно находиться в диапазоне от −20 до +19, приоритеты всегда попадают в промежуток от 1 до 40. Цель

алгоритма планирования состоит в том, чтобы обеспечить грубое пропорциональное соответствие качества обслуживания приоритету, то есть чем выше приоритет, тем меньше должно быть время отклика и тем большая доля процессорного времени достанется процессу.

Помимо приоритета с каждым процессом связан квант времени, то есть количество тиков таймера, в течение которых процесс может выполняться. По умолчанию системные часы тикают с частотой 100 Гц, так что каждый тик равен 10 мс. Этот интервал в системе Linux называют **«джиффи»** (jiffy — мгновение, миг, момент). Планировщик использует приоритет и квант следующим образом. Сначала он вычисляет называемую в системе Linux **«добродетелью»** (goodness) величину каждого готового процесса по следующему алгоритму:

```
if (class == real_time) goodness = 1000 + priority;
if (class == timesharing && quantum > 0) goodness = quantum + priority;
if (class == timesharing && quantum == 0) goodness = 0;
```

Для обоих классов реального времени выполняется первое условие. Все, что дает пометка процесса, как процесса реального времени, — это гарантия, что этот процесс получит более высокое значение *goodness*, чем все процессы разделения времени. У алгоритма есть еще одно дополнительное свойство: если у процесса, который запускался последним, осталось неиспользованное процессорное время, он получает бонус, позволяющий выиграть в спорных ситуациях. Идея состоит в том, что при прочих равных условиях более эффективным представляется запустить предыдущий процесс, так как его страницы и кэш с большой вероятностью еще находятся на своих местах.

В остальном алгоритм планирования очень прост: когда нужно принять решение, выбирается поток с максимальным значением «добродетели». Во время работы процесса его квант (переменная *quantum*) уменьшается на единицу на каждом тике. Центральный процессор отнимается у потока при выполнении одного из следующих условий:

1. Квант потока уменьшился до 0.
2. Поток блокируется на операции ввода-вывода, семафоре и т. д.
3. В состояние готовности перешел ранее заблокированный поток с более высокой «добродетелью».

Так как кванты постоянно уменьшаются, рано или поздно у любого потока квант станет нулевым. Однако у потока, блокированного вводом-выводом, может остаться некая ненулевая величина кванта. В этот момент планировщик пересчитывает значения квантов для *всех* потоков, как готовых, так и блокированных, по следующей формуле:

```
quantum = (quantum/2) + priority
```

где квант измеряется в «джиффи», то есть в тиках. Поток, ограниченный производительностью центрального процессора, как правило, быстро истратит свой квант, и при пересчете кванта его новое значение будет равно приоритету потока. В то же время у потока, ограниченного вводом-выводом, может остаться значительное количество неистраченного процессорного времени, поэтому в следующий раз значение его нового кванта будет больше, чем у потока, ограниченного производи-

тельностью процессора. Если системный вызов nice не используется, приоритет потока будет равен 20 и квант станет равным 20 тикам или 200 мс. С другой стороны, у потока, сильно ограниченного вводом-выводом, к моменту пересчета квантов может остаться квант, равный 20. Поэтому если его приоритет также равен 20, то новое значение его кванта будет равно 20/2 + 20 = 30 тиков. Если он опять заблокируется вводом-выводом, прежде чем успеет истратить один тик, то в следующий раз его квант будет равен 30/2 + 20 = 35 тиков. Эта величина стремится снизу к удвоенному значению приоритета. В результате применения данного алгоритма потоки, ограниченные вводом-выводом, получают большие кванты времени и, следовательно, считаются более «добродетельными», чем потоки, ограниченные производительностью процессора. Таким образом, потоки, ограниченные вводом-выводом, получают преимущество при планировании.

Другое свойство этого алгоритма заключается в том, что когда потоки, ограниченные производительностью процессора, соревнуются за право использования процессора, поток с большим приоритетом получает большую долю процессорного времени. В качестве примера рассмотрим два потока, ограниченных производительностью процессора: поток *A* с приоритетом 20 и поток *B* с приоритетом 5. Поток *A* запускается первым, и через 20 тиков его квант истекает. Затем запускается поток *B*, которому разрешается работать в течение 5 тиков. После 5 тиков, так как все кванты упали до нуля, они пересчитываются. Поток *A* снова получает 20 тиков, а поток *B* — 5 тиков. Так продолжается, пока один из потоков не выполнит всю свою работу, таким образом, поток *A* получает 80 % процессорного времени, а поток *B* получает 20 % процессорного времени.

Загрузка UNIX

Точные детали процесса загрузки операционной системы UNIX варьируются от системы к системе. Ниже будет кратко рассмотрено, как загружается 4.4BSD, но в своей основе это описание применимо и для других версий. Когда компьютер включается, в память считывается и исполняется первый сектор (главная загрузочная запись) загружаемого диска. Этот сектор содержит небольшую (512-байтовую) программу, загружающую автономную программу под названием *boot* с загрузочного устройства, как правило, с IDE или SCSI-диска. Программа *boot* сначала копирует саму себя в фиксированный адрес памяти в старших адресах, чтобы освободить нижнюю память для операционной системы.

Загрузившись, программа *boot* считывает корневой каталог с загрузочного устройства. Чтобы сделать это, она должна понимать формат файловой системы и каталога. Затем она считывает ядро операционной системы и передает ему управление. На этом программа *boot* завершает свою работу, после чего уже работает ядро системы.

Начальная программа ядра написана на ассемблере и является в значительной мере машинно-зависимой. Как правило, эта программа устанавливает указатель стека, определяет тип центрального процессора, вычисляет количество имеющегося в наличии ОЗУ, запрет прерываний, разрешение работы диспетчера памяти

и, наконец, вызывает процедуру *main*, написанную на С, чтобы запустить основную часть операционной системы.

Программа на языке С также должна проделать значительную работу по инициализации, но эта инициализация скорее логическая, нежели физическая. Она начинается с того, что выделяет память под буфер сообщений, что должно помочь решению проблем с загрузкой системы. По мере выполнения инициализации в этот буфер записываются сообщения, информирующие о том, что происходит в системе. В случае неудачной загрузки их можно выудить оттуда с помощью специальной программы диагностики. Этот буфер подобен черному ящику, который обычно пытаются найти на месте крушения самолета.

Затем выделяется память для структур данных ядра. Большинство этих структур имеют фиксированный размер, но размер некоторых из них, например размер буферного кэша и некоторых структур таблиц управления страницами памяти, зависит от доступного объема оперативной памяти.

Затем операционная система начинает определение конфигурации компьютера. Операционная система считывает файлы конфигурации, в которых сообщается, какие типы устройств ввода-вывода могут присутствовать, и проверяет, какие из устройств действительно присутствуют. Если проверяемое устройство отвечает, оно добавляется к таблице подключенных устройств. Если устройство не отвечает, оно считается отсутствующим и в дальнейшем игнорируется.

Как только список устройств определен, операционная система должна найти драйверы устройств. В этом месте версии UNIX несколько различаются между собой. В частности, система 4.4BSD не может динамически загружать драйверы устройств, поэтому любое устройство ввода-вывода, чей драйвер не был статически скомпонован с ядром, не может использоваться. Некоторые другие версии UNIX, как, например, Linux, напротив, могут динамически загружать драйверы (как это могут делать все версии MS-DOS и Windows).

Аргументы в пользу динамической загрузки драйверов и против нее весьма интересны и их стоит кратко упомянуть. Главный аргумент в пользу динамической загрузки заключается в том, что клиентам с различными конфигурациями может быть поставлен один и тот же двоичный файл, который автоматически загрузит необходимые ему драйверы, возможно даже по сети. Главный аргумент против динамической загрузки состоит в том, что этот метод противоречит принципам безопасности системы. Если вы управляете защищенным сайтом, например базой данных банка или корпоративным web-сервером, вероятно, вы захотите запретить кому бы то ни было вставлять случайные программы в ядро операционной системы. Системный администратор может хранить исходные тексты операционной системы и объектные файлы на защищенной машине и выполнять все работы по трансляции и компоновки системы на ней, после чего переносить двоичный код ядра на другие машины по локальной сети. Если драйверы не могут загружаться динамически, такой сценарий предотвращает установку в ядро не отлаженной или реализующей чьи-либо злые намерения программы системными операторами или еще кем-либо, кому известен пароль суперпользователя. Более того, в больших системах конфигурация аппаратуры точно известна уже во время компиляции и компоновки операционной системы. Изменения производятся довольно редко,

поэтому перекомпоновка системы при добавлении нового устройства не представляет собой проблемы.

Итак, когда загружающаяся операционная система определила конфигурацию аппаратного обеспечения, она должна аккуратно загрузить процесс 0, установить его стек и запустить этот процесс. Процесс 0 продолжает инициализацию, выполняя такие задачи, как программирование таймера реального времени, монтирование корневой файловой системы и создание процесса 1 (*init*) и страничного демона (процесс 2).

Процесс *init* проверяет свои флаги, в зависимости от которых он запускает операционную систему либо в однопользовательском, либо в многопользовательском режиме. В первом случае он создает процесс, выполняющий оболочку, и ждет, когда тот завершит свою работу. Во втором случае процесс *init* создает процесс, исполняющий сценарий оболочки инициализации системы */etc/rc*, который может выполнять проверку непротиворечивости файловой системы, монтировать дополнительные файловые системы, запускать демонов и т. д. Затем он считывает файл */etc/ttys*, в котором перечисляются терминалы и некоторые их свойства. Для каждого разрешенного терминала он создает копию самого себя, которая затем исполняет программу *getty*.

Программа *getty* устанавливает для каждой линии (некоторые из них могут быть, например, модемами) скорость линии, после чего выводит на терминале приглашение к входу в систему:

```
login:
```

После этого программа *getty* пытается прочитать имя пользователя, введенное с клавиатуры. Когда пользователь садится за терминал и вводит свое имя, программа *getty* завершает свою работу выполнением программы регистрации */bin/login*. После этого программа *login* запрашивает у пользователя его пароль, зашифровывает его и сравнивает с зашифрованным паролем, хранящимся в файле паролей */etc/passwd*. Если пароль введен верно, программа *login* вместо себя запускает оболочку пользователя, которая ждет первой команды. Если пароль введен неверно, программа *login* просто еще раз спрашивает имя пользователя. Этот механизм проиллюстрирован на рис. 10.5 для системы с тремя терминалами.

На рисунке процесс *getty*, работающий на терминале 0, все еще ждет ввода. На терминале 1 пользователь ввел имя регистрации, поэтому программа *getty* запустила поверх себя процесс *login*, запрашивающий пароль. На терминале 2 уже прошла успешная регистрация, в результате чего оболочка напечатала приглашение к вводу (%). Пользователь ввел команду

```
cp f1 f2
```

в результате которой оболочка создала дочерний процесс, исполняющий программу *cp*. Процесс оболочки блокируется в ожидании завершения дочернего процесса, после чего оболочка снова напечатает приглашение к вводу и будет ждать ввода с клавиатуры следующей команды. Если бы пользователь на терминале 2 вместо *cp* ввел *cc*, то запустилась бы главная программа компилятора С, который, в свою очередь, запустил бы несколько дочерних процессов для выполнения различных проходов компилятора.

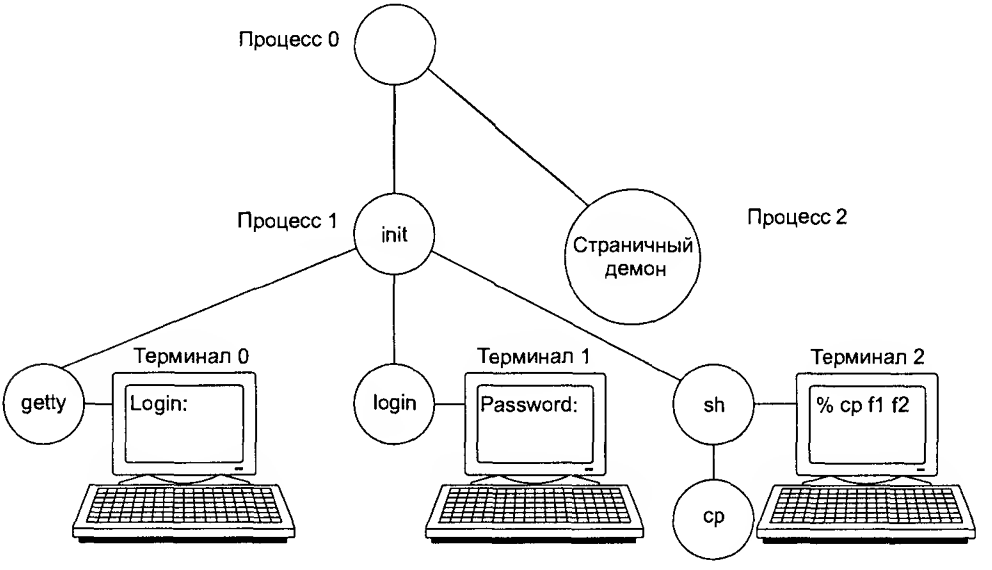


Рис. 10.5. Последовательность исполняемых процессов при загрузке некоторых версий системы UNIX

Управление памятью в UNIX

Модель памяти, используемая в системе UNIX, довольно проста, что должно обеспечить переносимость программ, а также реализацию операционной системы UNIX на машинах с сильно отличающимися модулями памяти, варьирующимися от элементарных (например, оригинальная IBM PC) до сложного оборудования со страничной организацией. Эта область практически не изменилась за последние несколько десятков лет. Разработанные уже давно архитектурные решения хорошо себя зарекомендовали и не требуют серьезной переработки. Мы рассмотрим модель управления памятью в системе UNIX и методы ее реализации.

Основные понятия

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста (программы), данных и стека. Пример адресного пространства процесса изображен на рис. 10.6, *а*. **Текстовый (программный) сегмент** содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы, написанной на языке высокого уровня (например, C или C++) в машинный код. Как правило, текстовый сегмент разрешен только для чтения. Самомодифицирующиеся программы вышли из моды примерно в 1950 году, так как их было слишком сложно понимать и отлаживать. Таким образом, текстовый сегмент не изменяется ни в размерах, ни по своему содержанию.

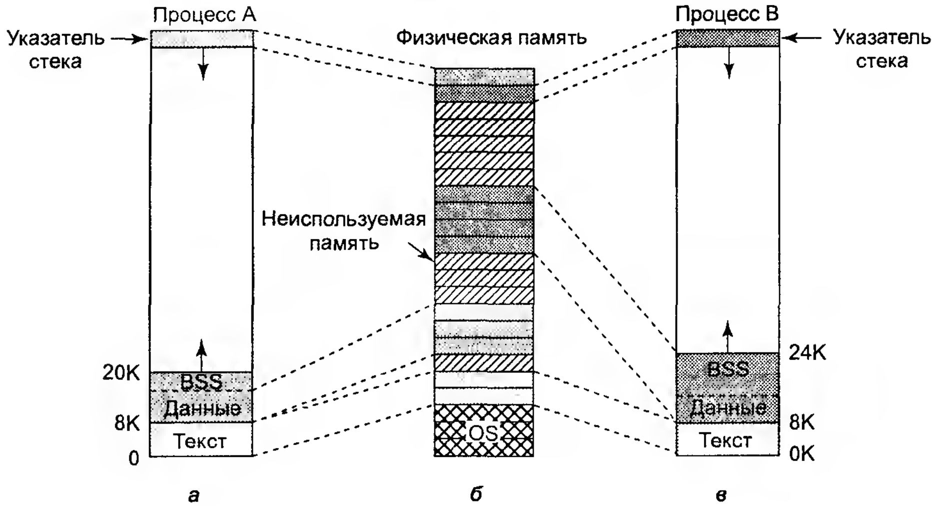


Рис. 10.6. Виртуальное адресное пространство процесса *A* (а); физическая память (*б*); виртуальное адресное пространство процесса *B* (в)

Сегмент данных содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных данных и неинициализированных данных. По историческим причинам вторая часть называется **BSS** (Bulk Storage System — запоминающее устройство большой емкости, массовое ЗУ). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы.

Например, на языке C можно объявить символьную строку и в то же время задать ее значение, то есть проинициализировать ее. Когда программа запускается, она предполагает, что в этой строке уже содержится некий осмысленный текст. Чтобы реализовать это, компилятор назначает строке определенное место в адресном пространстве и гарантирует, что в момент запуска программы по этому адресу будет располагаться соответствующая строка. С точки зрения операционной системы, инициализированные данные не отличаются от текста программы — и тот и другой сегменты содержат сформированные компилятором последовательности битов, загружаемые в память при запуске программы.

Неинициализированные данные необходимы лишь с точки зрения оптимизации. Когда начальное значение глобальной переменной явно не указано, то, согласно семантике языка C, ее значение устанавливается равным 0. На практике большинство глобальных переменных не инициализируются, и, таким образом, их начальное значение равно 0. Это можно реализовать следующим образом: создать целый сегмент исполняемого двоичного файла, точно равного по размеру числу байтов данных, и проинициализировать весь этот сегмент нулями.

Однако из экономии места на диске этого не делается. Файл содержит только те переменные, начальные значения которых явно заданы. Вместо неинициализированных переменных компилятор помещает в исполняемый файл просто одно слово, содержащее размер области неинициализированных данных в байтах. При запуске программы операционная система считывает это слово, выделяет нужное число байтов и обнуляет их.

Рассмотрим это еще раз на нашем примере (см. рис. 10.6, *а*). Здесь текст программы занимает 8 Кбайт, и инициализированные данные также занимают 8 Кбайт. Размер сегмента неинициализированных данных (BSS) равен 4 Кбайт. Исполняемый файл содержит только 16 Кбайт (текст + инициализированные данные), плюс короткий заголовок, в котором операционной системе указывается выделить программе дополнительно 4 Кбайт после неинициализированных данных и обнулить их перед выполнением программы. Этот трюк позволяет сэкономить 4 Кбайт нулей на диске в исполняемом файле.

В отличие от текстового сегмента, который не может изменяться, сегмент данных может модифицироваться. Программы изменяют свои переменные постоянно. Более того, многим программам требуется выделение дополнительной памяти динамически, во время выполнения. Чтобы реализовать это, операционная система UNIX разрешает сегменту данных расти при динамическом выделении памяти программам и уменьшаться при освобождении памяти программами. Программа может установить размер своего сегмента данных с помощью системного вызова brk. Таким образом, чтобы получить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом пользуется библиотечная процедура *malloc*, используемая для выделения памяти.

Третий сегмент — это сегмент стека. На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и растет вниз к 0. Если указатель стека оказывается ниже нижней границы сегмента стека, как правило, происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу памяти. Программы не управляют явно размером сегмента стека.

Когда программа запускается, ее стек не пуст. Напротив, он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке при вызове этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена. Например, когда вводится команда

```
cp src dest
```

запускается программа *cp* со строкой «cp src dest» в стеке, что позволяет ей определить имена файлов, с которыми ей предстоит работать. Строка представляется в виде массива указателей на отдельные аргументы командной строки, что облегчает ее обработку.

Когда два пользователя запускают одну и ту же программу, например текстовый редактор, в памяти можно хранить две копии программы редактора. Однако такой подход является неэффективным. Вместо этого большинством систем UNIX поддерживаются **текстовые сегменты совместного использования**. На рис. 10.6, *б* и *в* мы видим два процесса, *А* и *В*, совместно использующие общий текстовый сегмент. Отображение выполняется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова fork, и то только те страницы, которые не модифицируются любым из процессов. Если размер любого из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, так как соседние виртуальные страницы памяти не обязаны отображаться на соседние физические страницы.

На некоторых компьютерах аппаратное обеспечение поддерживает раздельные адресные пространства для команд и для данных. Если такая возможность есть, система UNIX может ею воспользоваться. Например, на компьютере с 32-разрядными адресами при возможности использования раздельных адресных пространств можно получить 2^{32} бит[1] адресного пространства для команд и еще 2^{32} бит адресного пространства для данных. Передача управления по адресу 0 будет восприниматься как передача управления по адресу 0 в текстовом пространстве, тогда как при обращении к данным по адресу 0 будет использоваться адрес 0 в пространстве данных. Таким образом, это свойство удваивает доступное адресное пространство.

Многими версиями UNIX поддерживается **отображение файлов на адресное пространство памяти**. Это свойство позволяет отображать файл на часть адресного пространства процесса, так чтобы можно было читать из файла и писать в файл, как если бы это был массив, хранящийся в памяти. Отображение файла на адресное пространство памяти делает произвольный доступ к нему существенно более легким, нежели при использовании системных вызовов, таких как `read` и `write`. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рис. 10.7 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

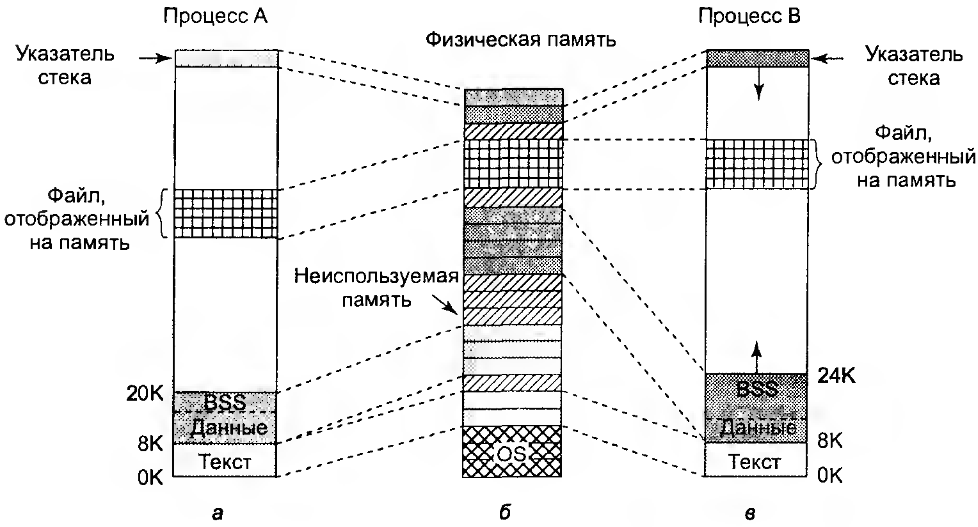


Рис. 10.7. Два процесса совместно используют один отображенный на память файл

Дополнительное преимущество отображения файла на память заключается в том, что два или более процессов могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным. Таким образом, отображение на адресное пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей

[1] Точнее, 2^{32} байт, что равно 4 Гбайт, так как к отдельным битам процессоры, как правило, адресуются внутри байта или слова. — *Примеч. перев.*

памяти для нескольких процессов, причем у такого механизма будет высокая пропускная способность. В предельном случае два или более процессов могут отобразить на память файл, покрывающий все адресное пространство, получая, таким образом, форму совместного использования памяти, что-то среднее между процессами и потоками. В этом случае, как и у потоков, все адресное пространство используется совместно, но каждый процесс может управлять собственными файлами и сигналами, что отличает этот вариант от потоков. Однако на практике такое никогда не применяется.

Системные вызовы управления памятью в UNIX

Стандартом POSIX системные вызовы для управления памятью не определяются. Эту область посчитали слишком машинно-зависимой, чтобы ее стандартизировать. Вместо этого просто сделали вид, что проблемы не существует, и заявили, что программы, которым требуется динамическое управление памятью, могут использовать библиотечную процедуру *malloc* (определенную стандартом ANSI C). Таким образом, вопрос реализации процедуры *malloc* был вынесен за пределы рассмотрения стандарта POSIX. В некоторых кругах такой подход был расценен как перекладывание бремени решения проблемы на чужие плечи.

На практике в большинстве систем UNIX есть системные вызовы для управления памятью. Наиболее распространенные системные вызовы перечислены в табл. 10.6. В случае ошибки возвращаемое значение *s* равно –1; переменные *a* и *addr* представляют собой адреса в памяти, *len* — это длина, параметр *prot* управляет защитой, *flags* содержит различные биты, *fd* — это дескриптор файла, а *offset* — смещение. Системный вызов brk указывает размер сегмента данных, задавая адрес первого байта за его пределами. Если новое значение больше старого, сегмент данных увеличивается, в противном случае он уменьшается.

Таблица 10.6. Некоторые системные вызовы для управления памятью

Системный вызов	Описание
s=brk(addr)	Изменить размер сегмента данных
a=mmap(addr, len, prot, flags, fd, offset)	Отобразить файл на память
s=unmap(addr, len)	Отменить отображения файла на память

Системные вызовы mmap и unmap управляют отображением файлов на адресное пространство памяти. Первый параметр системного вызова mmap, *addr*, указывает адрес, по которому будет отображаться файл (или его часть). Он должен быть кратен размеру страницы. Если этот параметр равен 0, тогда операционная система определяет этот адрес сама и возвращает его в *a*. Второй параметр, *len*, задает количество отображаемых байтов. Он также должен быть кратен размеру страницы. Третий параметр, *prot*, задает режим защиты для отображаемого файла. Файл может быть помечен как доступный для чтения, записи, исполнения или любой комбинации этих трех битов. Четвертый параметр, *flags*, определяет, является ли отображаемый файл приватным или доступным для совместного использования, а также содержит ли параметр *addr* жесткое требование или это всего лишь намек. Пятый параметр,

fd, представляет собой дескриптор отображаемого файла. Отображаться могут только открытые файлы. Наконец, параметр *offset* сообщает, с какого места должен отображаться файл. Файл может быть отображен, начиная с любого байта.

Второй системный вызов, unmap, отменяет отображения файла на память. Если отменяется отображение только части файла, то остальная часть файла продолжает отображаться на память.

Реализация управления памятью в UNIX

До версии 3BSD большинство систем UNIX основывались на свопинге (подкачке), работавшем следующим образом. Когда загружалось больше процессов, чем могло поместиться в памяти, некоторые из них выгружались на диск. Выгружаемый процесс всегда выгружался на диск целиком (исключение представляли только совместно используемые текстовые сегменты). Таким образом, процесс мог быть либо в памяти, либо на диске.

Свопинг

Перемещением данных между памятью и диском управлял верхний уровень двухуровневого планировщика, называвшийся **свопером** (swapper). Выгрузка данных из памяти на диск инициировалась, когда у ядра кончалась свободная память из-за одного из следующих событий:

1. Системному вызову fork требовалась память для дочернего процесса.
2. Системный вызов brk собирался расширить сегмент данных.
3. Разросшемуся стеку требовалась дополнительная память.

Кроме того, когда наступало время запустить процесс, уже достаточно долго находящийся на диске, часто бывало необходимо удалить из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Выбирая жертву, свопер сначала рассматривал блокированные (например, ожиданием ввода с терминала) процессы. Лучше удалить из памяти процесс, который не может работать, чем работоспособный процесс. Если такие процессы находились, из них выбирался процесс с наивысшим значением суммы приоритета и времени пребывания в памяти. Таким образом, хорошими кандидатами на выгрузку были процессы, потребившие большое количество процессорного времени, либо находящиеся в памяти уже достаточно долгое время, даже если большую его часть они занимались вводом-выводом. Если блокированных процессов не было, тогда на основе тех же критериев выбирался готовый процесс.

Каждые несколько секунд свопер исследовал список выгруженных процессов, проверяя, не готов ли какой-либо из этих процессов к работе. Если процессы в состоянии готовности обнаруживались, из них выбирался процесс, дольше всех находящийся на диске. Затем свопер проверял, будет ли это легкий свопинг или тяжелый. Легким свопингом считался тот, для которого не требовалось дополнительное высвобождение памяти. При этом нужно было всего лишь загрузить выгруженный на диск процесс. Тяжелым свопингом назывался свопинг, при котором для загрузки в память выгруженного на диск процесса из нее требовалось удалить один или несколько других процессов.

Затем весь этот алгоритм повторялся до тех пор, пока не выполнялось одно из следующих двух условий: (1) на диске не оставалось процессов, готовых к работе, или (2) в памяти не оставалось места для новых процессов. Чтобы не терять большую часть производительности системы на свопинг, ни один процесс не выгружался на диск, если он пробыл в памяти менее 2 с.

Свободное место в памяти и на устройстве перекачки учитывалось при помощи связного списка свободных пространств. Когда требовалось свободное пространство в памяти или на диске, из списка выбиралось первое подходящее свободное пространство. После этого в список возвращался остаток от свободного пространства.

Постраничная подкачка в системе UNIX

Все версии операционной системы UNIX для компьютеров PDP-11 и Interdata, а также начальная версия для машины VAX были основаны на свопинге, о котором только что рассказывалось. Однако, начиная с версии 3BSD, университет в Беркли добавил к системе страничную подкачку, чтобы предоставить возможность работать с программами самых больших размеров. Практически во всех версиях системы UNIX теперь есть страничная подкачка по требованию, появившаяся впервые в версии 3BSD. Ниже мы опишем строение версии 4BSD, но System V основана на 4BSD и во многом схожа с нею.

Идея, лежащая в основе страничной подкачки в системе 4BSD, проста: чтобы работать, процессу не нужно целиком находиться в памяти. Все, что в действительности требуется, — это структура пользователя и таблицы страниц. Если они загружены, то процесс считается находящимся в памяти и может быть запущен планировщиком. Страницы с сегментами текста, данных и стека загружаются в память динамически, по мере обращения к ним. Если пользовательской структуры и таблицы страниц нет в памяти, то процесс не может быть запущен, пока свопер не загрузит их.

В системе Berkeley UNIX не используется модель рабочего набора или любая другая форма опережающей подкачки страниц, так как для этого требуется знать, какие страницы используются в данный момент, а какие нет. Поскольку в машине VAX не было битов обращений к памяти, эту информацию было получить непросто (хотя это можно было поддержать программно за счет значительных дополнительных накладных расходов).

Страничная подкачка реализуется частично ядром и частично новым процессом, называемым **страничным демоном**. Страничный демон — это процесс 2 (процесс 0 — это свопер, а процесс 1 — *init*, как показано на рис. 10.5). Как и все демоны, страничный демон периодически запускается и смотрит, есть ли для него работа. Если он обнаруживает, что количество страниц в списке свободных страниц слишком мало, страничный демон инициирует действия по освобождению дополнительных страниц.

Организация памяти в 4BSD показана на рис. 10.8. Память делится на три части. Первые две части, ядро операционной системы и карта памяти, фиксированы в физической памяти (то есть никогда не выгружаются). Остальная память компьютера делится на страничные блоки, каждый из которых может содержать страницу текста, данных или стека или находиться в списке свободных страниц.

Карта памяти содержит информацию о содержимом страничных блоков. Для каждого страничного блока в карте памяти есть запись фиксированной длины. При килобайтных страничных блоках и 16-байтовых записях карты памяти на карту памяти расходуется менее 2 % от общего объема памяти. Первые два поля записи карты памяти используются только тогда, когда соответствующий страничный блок находится в списке свободных страниц. В этом случае они сшивают свободные страницы в двусвязный список. Следующие три записи используются, когда страничный блок содержит информацию. У каждой страницы в памяти есть фиксированное место хранения на диске, в которое она помещается, когда выгружается из памяти. Еще три поля содержат ссылку на запись в таблице процессов, тип хранящегося в странице сегмента и смещение в сегменте процесса. Последнее поле содержит некоторые флаги, нужные для алгоритма страничной подкачки.

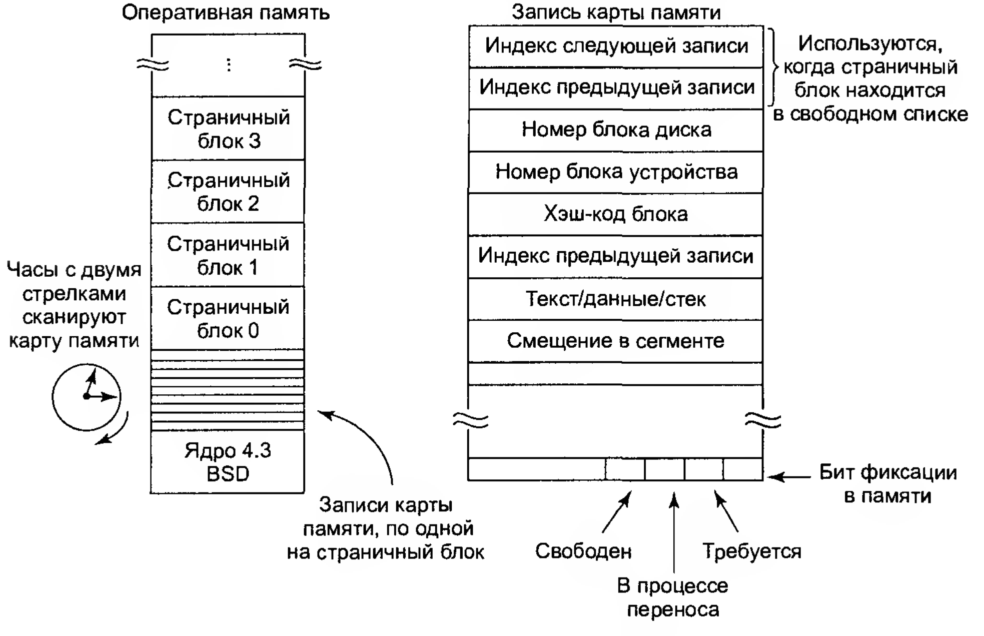


Рис. 10.8. Карта памяти в 4BSD

При запуске процесс может вызвать страничное прерывание, если одной или нескольких его страниц не окажется в памяти. При страничном прерывании операционная система берет первый страничный блок из списка свободных страниц, удаляет его из списка и считывает в него требуемую страницу. Если список свободных страниц пуст, выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит страничный блок.

Алгоритм замещения страниц

Алгоритм замещения страниц выполняется страничным демоном. Раз в 250 мс он просыпается, чтобы сравнить количество свободных страничных блоков с системным параметром *lotsfree* (равным, как правило, 1/4 объема памяти). Если число

свободных страничных блоков меньше, чем значение этого параметра, страничный демон начинает переносить страницы из памяти на диск, пока количество свободных страничных блоков не станет равно *lotsfree*. Если же количество свободных страничных блоков больше или равно *lotsfree*, тогда страничный демон, ничего не предпринимая, отправляется спать дальше. Если у компьютера много памяти и мало активных процессов, страничный демон спит практически все время.

Страничный демон использует модифицированную версию алгоритма часов. Это глобальный алгоритм, то есть при удалении страницы он не учитывает, чья это страница. Таким образом, количество страниц, выделяемых каждому процессу, меняется со временем.

Основной алгоритм часов работает, сканируя в цикле страничные блоки (как если бы они лежали на окружности циферблата часов). На первом проходе, когда стрелка часов указывает на страничный блок, сбрасывается его бит использования. На втором проходе у каждого страничного блока, к которому не было доступа с момента первого прохода, бит использования останется сброшенным, и этот страничный блок будет помещен в список свободных страниц (после записи его на диск, если он «грязный»). Страничный блок в списке свободных страниц сохраняет свое содержание, что позволяет восстановить страницу, если она потребуется прежде, чем будет перезаписана.

На машинах типа VAX, у которых не было битов использования, когда стрелка часов указывала на страничный блок на первом проходе, сбрасывался программный бит, а страница помечалась в таблице страниц как недействительная. При следующем доступе к странице происходило страничное прерывание, что позволяло операционной системе установить программный бит использования. Эффект достигался тот же самый, что и при использовании аппаратного бита использования, но реализация была значительно более сложной и медленной. Таким образом, программное обеспечение расплачивалось за недостаточно развитую схему аппаратного обеспечения.

Изначально в Berkley UNIX использовался основной алгоритм часов, но затем было обнаружено, что при больших объемах оперативной памяти проходы занимают слишком много времени. Тогда алгоритм был заменен **алгоритмом часов с двумя стрелками** (см. рис. 10.8). В этом алгоритме страничный демон поддерживает два указателя на карту памяти. При работе он сначала очищает бит использования передней стрелкой, а затем проверяет этот бит задней стрелкой. После чего перемещает обе стрелки. Если две стрелки находятся близко друг от друга, то только у очень активно используемых страниц появляется шанс, что к ним будет обращение между проходами двух стрелок. Если же стрелки разнесены на 359 градусов (то есть задняя стрелка находится слегка впереди передней), мы снова получаем исходный алгоритм часов. При каждом запуске страничного демона стрелки проходят не полный оборот, а столько, сколько необходимо, чтобы количество страниц в списке свободных страниц было не менее *lotsfree*.

Если операционная система обнаруживает, что частота подкачки страниц слишком высока, а количество свободных страниц все время ниже *lotsfree*, свопер начинает удалять из памяти один или несколько процессов, чтобы остановить состязание за свободные страничные блоки. Алгоритм свопинга в системе 4BSD следующий. Сначала свопер проверяет, есть ли процесс, который бездействовал

в течение 20 и более секунд. Если такие процессы есть, из них выбирается бездействовавший в течение максимального срока и выгружается на диск. Если таких процессов нет, изучаются четыре самых больших процесса, из которых выбирается тот, который находился в памяти дольше всех, и выгружается на диск. При необходимости этот алгоритм повторяется до тех пор, пока не будет высвобождено достаточное количество памяти.

Каждые несколько секунд свопер проверяет, есть ли на диске готовые процессы, которые следует загрузить в память. Каждому процессу на диске присваивается значение, зависящее от времени его пребывания в выгруженном состоянии, размера, значения, использовавшегося при обращении к системному вызову nice (если такое обращение было), и от того, как долго этот процесс спал, прежде чем был выгружен на диск. Эта функция обычно взвешивается так, чтобы загружать в память процесс, дольше всех находящийся в выгруженном состоянии, если только он не крайне большой. Теория утверждает, что загружать большие процессы дорого, поэтому их не следует перемещать туда-сюда слишком часто. Загрузка процесса производится только при условии наличия достаточного количества свободных страниц, чтобы, когда случится неизбежное страничное прерывание, для него нашлись свободные страничные блоки. Свопер загружает в память только структуру пользователя и таблицы страниц. Страницы с текстом, данными и стеком подгружаются при помощи обычной страничной подкачки.

У каждого сегмента каждого активного процесса есть место на диске, где он располагается, когда его страницы удаляются из памяти. Сегменты данных и стека сохраняются на временном устройстве, но текст программы подгружается из самого исполняемого двоичного файла. Для текста программы временная копия не используется.

Страничная подкачка в System V во многом схожа с применяемой в системе 4BSD, что не удивительно, так как версия UNIX университета в Беркли уже в течение многих лет стабильно работала, прежде чем страничная подкачка была добавлена к System V. Тем не менее между этими версиями операционной системы есть два интересных различия.

Во-первых, в System V вместо алгоритма часов с двумя стрелками используется оригинальный алгоритм часов с одной стрелкой. Более того, вместо того чтобы помещать страницу в список свободных страниц на втором проходе, страница помещается туда только в случае, если она не использовалась в течение нескольких последовательных проходов. Хотя при таком решении страницы не освобождаются так быстро, как это делается алгоритмом в Berkeley UNIX, оно значительно увеличивает вероятность того, что освобожденная страница не потребуется тут же снова.

Во-вторых, вместо единственной переменной *lotsfree* в System V используются две переменные, *min* и *max*. Когда количество свободных страниц опускается ниже *min*, страничный демон начинает освобождать страницы. Демон продолжает работать до тех пор, пока число свободных страниц не сравняется со значением *max*. Такой подход позволяет избежать неустойчивости, возможной в системе 4DSD. Рассмотрим ситуацию, в которой количество свободных страниц на единицу меньше, чем *lotsfree*, поэтому страничный демон освобождает одну страницу, чтобы привести количество свободных страниц в соответствие со значением *lotsfree*. Затем происходит еще одно страничное прерывание, и количество свободных страниц

опять становится на единицу меньше *lotsfree*, в результате страничный демон снова начинает работать. Если установить значение *min* существенно большим, чем *min*, страничный демон, завершив освобождение страниц, создает достаточный запас, чтобы иметь возможность отдохнуть некоторое время.

Управление памятью в Linux

Каждый процесс системы Linux на 32-разрядной машине получает 3 Гбайт виртуального адресного пространства для себя, с оставшимся 1 Гбайт памяти для страничных таблиц и других данных ядра. Один гигабайт ядра не виден в пользовательском режиме, но становится доступным, когда процесс переключается в режим ядра. Адресное пространство создается при создании процесса и перезаписывается системным вызовом exec.

Виртуальное адресное пространство делится на однородные непрерывные области, выровненные по границам страниц. Таким образом, каждая область состоит из набора соседних страниц с одинаковым режимом защиты и одинаковыми свойствами подкачки. Примерами областей являются текстовый сегмент и файлы, отображенные на память (см. рис. 10.7). Между областями в виртуальном адресном пространстве могут быть свободные участки. Любое обращение процесса к памяти в этих свободных участках приводит к фатальному страничному прерыванию. Размер страницы фиксирован, например 4 Кбайт для процессора Pentium и 8 Кбайт для процессора Alpha.

Каждая область описывается в ядре записью *vm_area_struct*. Все структуры *vm_area_struct* одного процесса связаны вместе в список, отсортированный по виртуальным адресам, что позволяет быстро находить все страницы. Когда список становится слишком длинным (более 32 записей), создается дерево для ускорения поиска. Запись *vm_area_struct* перечисляет свойства области. К ним относятся режим защиты (например, только чтение или чтение/запись), является ли данная область фиксированной в памяти (невыгружаемой), и направление, в котором область может расти (вверх для сегментов данных, вниз для сегмента стека).

Структура *vm_area_struct* также содержит данные о том, является ли данная область приватной областью процесса или ее совместно используют несколько процессов. После системного вызова fork система Linux создает копию списка областей для дочернего процесса, но у дочернего и родительского процессов оказываются указатели на одни и те же таблицы страниц. Области помечаются как доступные для чтения/записи, но страницы доступны только для чтения. Если любой из процессов пытается записать данные в такую страницу, происходит прерывание, ядро видит, что область логически доступна для записи, а страница недоступна, поэтому оно дает процессу копию страницы, которую помечает как доступную для чтения/записи. Таким образом реализован механизм копирования при записи.

Кроме того, в структуре *vm_area_struct* записано, есть ли у этой области памяти место хранения на диске, и если да, то где оно расположено. Текстовые сегменты в качестве резервного хранения используют двоичные файлы, а отображаемые на адресное пространство памяти файлы выгружаются на диск в соответствующие им файлы. Всем остальным областям, таким как область стека, не назначаются области резервного хранения, пока не потребуется их выгрузка на диск.

В системе Linux используется трехуровневая схема страничной подкачки. Хотя эта схема была реализована в системе для процессора Alpha, она также используется (в упрощенном виде) для всех архитектур. Каждый виртуальный адрес разбивается на четыре поля, как показано на рис. 10.9. Каталоговое поле используется как индекс в глобальном каталоге, в котором есть личный каталог для каждого процесса. Содержание элемента каталога является указателем на одну из средних страничных таблиц, которые тоже проиндексированы полем виртуального адреса. Наконец, элемент средней таблицы указывает на таблицу страниц, также проиндексированную полем страницы виртуального адреса. Элемент в последней таблице содержит указатель на нужную страницу. На компьютерах с процессором Pentium используется только двухуровневая организация страниц. В этом случае каждый средний страничный каталог содержит только одну запись. Таким образом, глобальный каталог фактически указывает на таблицу страниц.

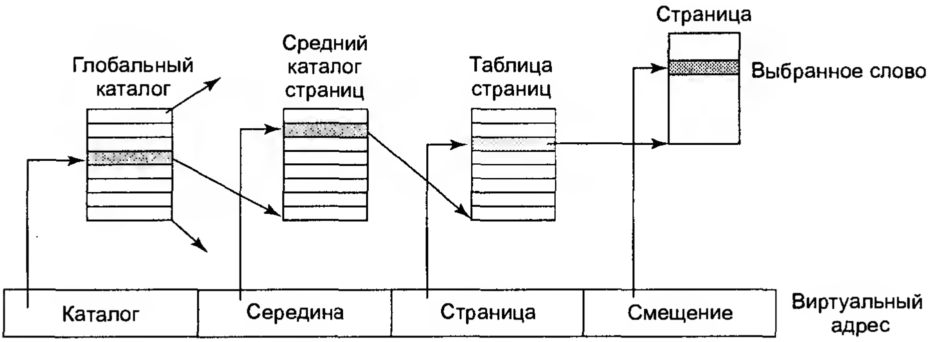


Рис. 10.9. В Linux используются трехуровневые таблицы страниц

Физическая память используется для различных целей. Само ядро жестко фиксировано. Ни одна его часть не выгружается на диск. Остальная часть памяти доступна для страниц пользователей, буферного кэша, используемого файловой системой, страничного кэша и других задач. Буферный кэш содержит блоки файлов, которые были недавно считаны или были считаны заранее в надежде на то, что они скоро могут понадобиться. Его размер динамически меняется. Буферный кэш состязается за место в памяти со страницами пользователей. Страничный кэш в действительности не является настоящим отдельным кэшем, а представляет собой просто набор страниц пользователя, которые более не нужны и ожидают выгрузки на диск. Если страница, находящаяся в страничном кэше, потребуется снова, прежде чем она будет удалена из памяти, ее можно быстро объявить находящейся в памяти.

Кроме этого, операционная система Linux поддерживает динамически загружаемые модули, в основном драйверы устройств. Они могут быть произвольного размера и каждому из них должен быть выделен непрерывный участок в памяти ядра. Для выполнения всех этих требований система Linux управляет памятью таким образом, что она может получить по желанию участок памяти произвольного размера. Для этого используется алгоритм, известный как **«дружественный» алгоритм**. Он будет описан ниже.

Основная идея управления блоками памяти заключается в следующем. Изначально память состоит из единого непрерывного участка. В нашем примере на рис. 10.10, *а* размер этого участка равен 64 страницам. Когда поступает запрос на выделение памяти, он сначала округляется до степени двух, например до 8 страниц. Затем весь блок памяти делится пополам (рис. 10.10, *б*). Так как получившиеся в результате этого деления надвое участки памяти все еще слишком велики, нижняя половина делится пополам еще (рис. 10.10, *в*) и еще (рис. 10.10, *г*). Теперь мы получили участок памяти нужного размера. Этот участок предоставляется обратившемуся процессу (затененный на рис. 10.10, *г*).

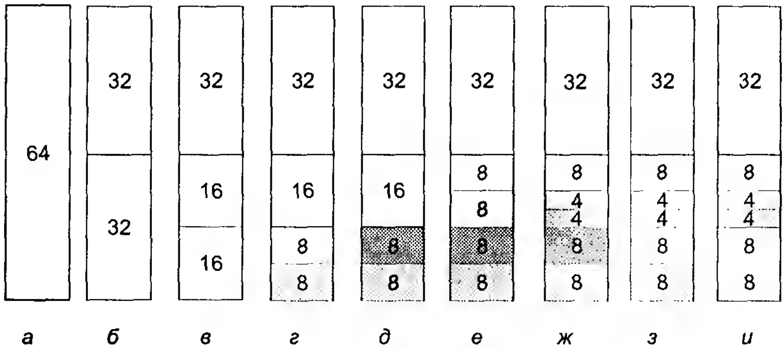


Рис. 10.10. Этапы работы дружественного алгоритма

Теперь предположим, что приходит второй запрос на 8 страниц. Он может быть удовлетворен немедленно (рис. 10.10, *д*). Следом поступает запрос на 4 страницы. При этом делится надвое наименьший участок (рис. 10.10, *е*) и выделяется половина от половины (рис. 10.10, *ж*). Затем освобождается второй 8-страничный участок (рис. 10.10, *з*). Наконец, освобождается оставшийся 8-страничный участок. Поскольку эти два участка были «приятелями», то есть они вышли из одного 16-страничного блока, они снова объединяются в 16-страничный блок (рис. 10.10, *и*).

Операционная система Linux управляет памятью при помощи данного алгоритма. К нему добавляется массив, в котором первый элемент представляет собой начало списка блоков размером в 1 единицу, второй элемент является началом списка блоков размером в 2 единицы, третий элемент — началом списка блоков размером в 4 единицы и т. д. Таким образом, можно быстро найти любой блок с размером кратным степени 2.

Этот алгоритм приводит к существенной внутренней фрагментации, так как, если вам нужен 65-страничный участок, вы получите 128-страничный блок.

Чтобы как-то решить эту проблему, в системе Linux есть второй алгоритм выделения памяти, выбирающий блоки памяти при помощи «приятельского» алгоритма, а затем нарезающий из этих блоков более мелкие фрагменты и управляющий этими фрагментами отдельно. Кроме того, существует третий алгоритм выделения памяти, использующийся, когда выделяемая память должна быть непрерывна только в виртуальном адресном пространстве, но не в физической памяти. Все эти алгоритмы выделения памяти были взяты из системы System V.

Операционная система Linux является системой, предоставляющей страницы по требованию, без предварительной загрузки страниц и без концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться). Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске. Все остальное выгружается либо в область подкачки, если она присутствует, либо в файлы подкачки фиксированной длины, которых может быть от одного до восьми. Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет. Выгрузка страниц в отдельный раздел диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем выгрузка в файл, по нескольким причинам. Во-первых, не требуется преобразование блоков файла в блоки диска. Во-вторых, физическая запись может быть любого размера, а не только размера блока файла. В-третьих, страница всегда пишется прямо на устройство в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не всегда так.

Страницы на устройстве подкачки или дисковом разделе подкачки не выделяются, пока они не потребуются. Каждое устройство или файл подкачки начинается с битового массива, в котором сообщается, какие страницы свободны. Когда страница, у которой нет места хранения на диске, должна быть удалена из памяти, из файлов (или разделов) подкачки, в которых еще есть свободное место, выбирается файл с наивысшим приоритетом, и в нем выделяется место для страницы. Как правило, у раздела подкачки (если таковой имеется) более высокий приоритет, чем у любого файла подкачки. Координата страницы на диске записывается в таблицу страниц.

Алгоритм замещения страниц работает следующим образом. Система Linux пытается поддерживать некоторые страницы свободными, чтобы их можно было предоставить при необходимости. Конечно, этот пул страниц должен постоянно пополняться, поэтому реальный алгоритм страничной подкачки заключается в том, как это происходит. Во время загрузки процесс *init* запускает страничный демон *kswapd*, который работает раз в секунду. Он проверяет, есть ли достаточное количество свободных страниц. Если да, он отправляется спать еще секунду, хотя он может быть разбужен и раньше, если внезапно понадобятся дополнительные страницы. Страничный демон состоит из цикла, который выполняется до шести раз с возрастающей срочностью. Почему шесть? Вероятно, автор программы думал, что четырех будет недостаточно, а восемь будет слишком много. В отдельных местах операционная система Linux реализована именно так.

Тело цикла выполняет обращения к трем процедурам, каждая из которых пытается получить различные типы страниц. Значение срочности передается в виде параметра, сообщающего процедуре, сколько усилий требуется предпринять, чтобы получить некоторые страницы. Как правило, это означает, сколько страниц нужно проверить, прежде чем опустить руки. В результате этот алгоритм сначала выбирает легко доступные страницы каждой категории, после чего переходит к труднодоступным. Когда получено достаточное количество страниц, страничный демон снова отправляется спать.

Первая процедура пытается получить те страницы из страничного кэша и буферного кэша файловой системы, к которым в последнее время не было обращений, для чего используется алгоритм часов. Вторая процедура ищет совместно исполь-

зуемые страницы, которыми никто из пользователей, похоже, не пользуется активно. Третья процедура, пытающаяся получить страницы, используемые одиночными пользователями, является наиболее интересной, поэтому рассмотрим ее подробнее.

Сначала выполняется цикл по всем процессам, в котором определяется, у какого процесса больше всего страниц на данный момент находится в памяти. Как только такой процесс найден, сканируются все его структуры *vm_area_struct* и изучаются все страницы в порядке виртуальных адресов, начиная с того места, на котором этот процесс был отложен в прошлый раз. Если страница недействительна, отсутствует в памяти, используется совместно, фиксирована в памяти или используется для DMA, то она пропускается. Если у страницы установлен бит обращения к ней, этот бит сбрасывается и страница отправляется в резерв. Если же бит сброшен, эта страница отнимается у процесса. В результате данный алгоритм подобен алгоритму часов (с той разницей, что страницы не сканируются в порядке FIFO).

Если страница, выбранная для удаления из памяти, чистая, она удаляется немедленно. Если страница «грязная» и у нее есть место резервного хранения на диске, она устанавливается в очередь записи на диск. Наконец, если у «грязной» страницы нет места резервного хранения на диске, она отправляется в страничный кэш, из которого она может быть снова получена позднее, если обращение к ней поступит прежде, чем она будет фактически выгружена на диск. В основе идеи сканирования страниц в порядке виртуальных адресов лежит надежда на то, что страницы, расположенные близко друг к другу в виртуальном адресном пространстве, скорее всего будут использоваться или не использоваться вместе, как единая группа, поэтому их следует записывать на диск как группу, а затем вместе считывать в память.

В управлении памятью принимает участие еще один демон, *bdflush*. Он периодически просыпается (а в некоторых случаях его явно будят), чтобы проверить, не превысило ли количество «грязных» страниц определенного предельного уровня. Если превысило, демон начинает сохранять их на диске.

Ввод-вывод в системе UNIX

Система ввода-вывода в UNIX довольно проста. Как правило, все устройства ввода-вывода выглядят как файлы, и доступ к ним осуществляется с помощью тех же системных вызовов `read` и `write`, которые используются для доступа к обычным файлам. В некоторых случаях должны быть заданы параметры устройства, для чего служит специальный системный вызов. В следующих разделах мы рассмотрим эти вопросы.

Основные понятия

Как и у всех компьютеров, у машин, работающих под управлением операционной системы UNIX, есть устройства ввода-вывода, такие как диски, принтеры и сети, соединенные с ними. Требуется некий способ предоставления программам доступа к этим устройствам. Хотя возможны различные варианты решений данного вопроса, подход, применяемый в операционной системе UNIX, заключается в интегри-

ровании всех устройств в файловую систему в виде так называемых **специальных файлов**. Каждому устройству ввода-вывода назначается имя пути, обычно в каталоге */dev*. Например, диск может иметь путь */dev/hd1*, у принтера может быть путь */dev/lp*, а у сети — */dev/net*.

Доступ к этим специальным файлам осуществляется так же, как и к обычным файлам. Для этого не требуется никаких специальных команд или системных вызовов. Вполне подойдут обычные системные вызовы `read` и `write`. Например, команда

```
cp file /dev/lp
```

скопирует файл *file* на принтер, в результате чего этот файл будет распечатан (при условии, что у пользователя есть разрешение доступа к */dev/lp*). Программы могут открывать, читать специальные файлы, а также писать в них тем же способом, что и в обычные файлы. На самом деле программа *cp* в приведенном выше примере даже не знает, что она занимается печатью файла на принтере. Таким образом, для выполнения ввода-вывода не требуется специального механизма.

Специальные файлы подразделяются на две категории: блочные и символьные. **Блочный специальный файл** — это специальный файл, состоящий из последовательности нумерованных блоков. Основное свойство блочного специального файла заключается в том, что к каждому его блоку можно адресоваться и получить доступ отдельно. Другими словами, программа может открыть блочный специальный файл и прочитать, скажем, 124-й блок, не читая сначала блоки с 0 по 123. Блочные специальные файлы обычно используются для дисков.

Символьные специальные файлы, как правило, используются для устройств ввода или вывода символьного потока. Символьные специальные файлы используются такими устройствами, как клавиатуры, принтеры, сети, мыши, плоттеры и т. д. Невозможно (и даже бессмысленно) искать на мыши 124-й блок.

С каждым специальным файлом связан драйвер устройства, осуществляющий управление соответствующим устройством. У каждого драйвера есть так называемый номер **старшего устройства**, служащий для его идентификации. Если драйвер одновременно поддерживает несколько устройств, например два диска одного типа, то каждому диску присваивается номер **младшего устройства**, идентифицирующий это устройство. Вместе номера главного устройства и младшего устройства однозначно обозначают каждое устройство ввода-вывода. В некоторых случаях один драйвер может управлять двумя связанными устройствами. Например, драйвер, соответствующий символьному специальному файлу */dev/tty*, управляет и клавиатурой и экраном, которые часто воспринимаются как единое устройство, терминал.

Хотя к большинству символьных специальных файлов невозможен произвольный доступ, ими часто бывает нужно управлять таким способом, который не используется для блочных специальных файлов. Рассмотрим, например, строку, введенную с клавиатуры и отображенную на экране. Когда пользователь по ошибке нажимает не ту клавишу и хочет заменить последний символ, он нажимает специальную клавишу. Некоторые пользователи предпочитают использовать для этого клавишу `BACKSPACE`, а другие любят пользоваться клавишей `DEL`. Для удаления всей только что набранной строки тоже имеется большой выбор средств. Традиционно использовался символ @, но с распространением электронной почты (исполь-

зующей символ @ в почтовом адресе) многие системы перешли на использование комбинации клавиш CTRL+U или других символов. Требуется и особая клавиша для прерывания работающей программы. Здесь также у разных пользователей различные вкусы.

В операционной системе UNIX эти символы не заданы жестко, а могут быть настроены пользователем. Для установки этих параметров обычно предоставляется специальный системный вызов. Этот системный вызов также управляет преобразованием табулятора в пробелы, включением и выключением эха при вводе, преобразованиями символов перевода каретки в перенос строки и т. д. Для обычных файлов и блочных специальных файлов этот системный вызов недоступен.

Работа с сетью

Другим примером ввода-вывода является работа с сетью, впервые появившаяся в Berkeley UNIX. Этот вопрос мы и рассмотрим ниже. Ключевым понятием в схеме Berkeley UNIX является **сокет**. Сокеты подобны почтовым ящикам и телефонным розеткам в том смысле, что они образуют пользовательский интерфейс с сетью, как почтовые ящики формируют интерфейс с почтовой системой, а телефонные розетки позволяют абоненту подключить телефон и соединиться с телефонной системой. Схематично расположение сокетов показано на рис. 10.11.

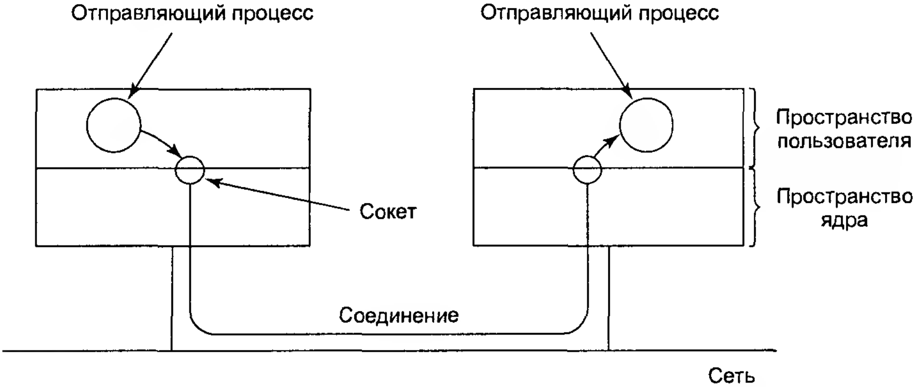


Рис. 10.11. Использование сокетов для соединения с сетью

Сокеты могут динамически создаваться и разрушаться. При создании сокета вызывающему процессу возвращается дескриптор файла, требующийся для установки соединения, чтения и записи данных, а также разрыва соединения.

Каждый сокет поддерживает определенный тип работы в сети, указываемый при создании сокета. Наиболее распространенными типами сокетов являются:

1. Надежный, ориентированный на соединение байтовый поток.
2. Надежный, ориентированный на соединение поток пакетов.
3. Ненадежная передача пакетов.

Первый тип сокетов позволяет двум процессам на различных машинах установить между собой эквивалент «трубы» (канала между процессами на одной маши-

не). Байты подаются в канал с одного конца и в том же порядке выходят с другого. Такая система гарантирует, что все посланные байты прибудут на другой конец канала и прибудут именно в том порядке, в котором были отправлены.

Второй тип сокетов отличается от первого тем, что он сохраняет границы между пакетами. Если отправитель пять раз отдельно обращается к системному вызову write, каждый раз отправляя по 512 байт, а получатель запрашивает 2560 байт по сокету типа 1, он получит все 2560 байт сразу. При использовании сокета типа 2 ему будут выданы только первые 512 байт. Чтобы получить остальные байты, получателю придется выполнить системный вызов read еще четыре раза. Третий тип сокета предоставляет пользователю доступ к «голой» сети. Этот тип сокета особенно полезен для приложений реального времени и ситуаций, в которых пользователь хочет реализовать специальную схему обработки ошибок. Сеть может терять пакеты или доставлять их в неверном порядке. В отличие от сокетов первых двух типов, сокет типа 3 не предоставляет никаких гарантий доставки. Преимущество этого режима заключается в более высокой производительности, которая в некоторых ситуациях оказывается важнее надежности (например, для доставки мультимедиа, при которой скорость ценится существенно выше, нежели сохранность данных по дороге).

При создании сокета один из параметров указывает протокол, используемый для него. Для надежных байтовых потоков, как правило, используется протокол **TCP** (Transmission Control Protocol — протокол управления передачей). Для ненадежной передачи пакетов обычно применяется протокол **UDP** (User Data Protocol — пользовательский протокол данных). Все эти протоколы были разработаны для сети ARPANET Министерства обороны США и теперь составляют основу Интернета. Для надежного потока пакетов специального протокола нет.

Прежде чем сокет может быть использован для работы в сети, с ним должен быть связан адрес. Этот адрес может принадлежать к одному из нескольких пространств адресов. Наиболее распространенным пространством является пространство адресов Интернета, использующее 32-разрядные числа для идентификации конечных адресатов в протоколе IP v 4 и 128-разрядные числа в протоколе IP v 6 (5-я версия протокола IP была экспериментальной системой, так и не выпущенной в свет).

Как только сокеты созданы на компьютере-источнике и компьютере-приемнике, между ними может быть установлено соединение (для ориентированной на соединение связи). Одна сторона обращается к системному вызову listen, указывая в качестве параметра локальный сокет. При этом системный вызов создает буфер и блокируется до тех пор, пока не прибудут данные. Другая сторона обращается к системному вызову connect, задавая в параметрах дескриптор файла для локального сокета и адрес удаленного сокета. Если удаленный компьютер принимает вызов, тогда система устанавливает соединение между двумя сокетами.

Функции установленного соединения аналогичны функциям канала. Процесс может читать из канала и писать в него, используя дескриптор файла для локального сокета. Когда соединение более не нужно, оно может быть закрыто обычным способом, при помощи системного вызова close.

Системные вызовы ввода-вывода системы UNIX

С каждым устройством ввода-вывода в операционной системе UNIX обычно связан специальный файл. Большую часть операций ввода-вывода можно выполнить при помощи соответствующего файла, что позволяет избежать необходимости использования специальных системных вызовов. Тем не менее иногда возникает необходимость в обращении к неким специфическим устройствам. До принятия стандарта POSIX в большинстве версий системы UNIX был системный вызов `ioctl`, выполнявший со специальными файлами большое количество операций, специфических для различных устройств. С годами все это привело к путанице. В стандарте POSIX этот вопрос был приведен в порядок, для чего функции системного вызова `ioctl` были разбиты на отдельные функциональные вызовы, главным образом для управления терминалом. Является ли каждый из них отдельным системным вызовом или все они вместе используют один системный вызов, зависит от конкретной реализации.

Первые четыре вызова, перечисленные в табл. 10.7, используются для задания скорости терминала. Для управления вводом и выводом предоставлены различные вызовы, так как некоторые модемы работают в несимметричном режиме. Например, старые системы videotex предоставляли пользователям доступ к открытым базам данных с короткими запросами от дома до сервера на скорости 75 бит/с и ответами, посылаемыми со скоростью 1200 бит/с. Этот стандарт был принят в то время, когда скорость 1200 бит/с в обоих направлениях была слишком дорогой для домашнего использования. Такая асимметрия все еще сохраняется на многих линиях связи. Например, некоторые телефонные компании предоставляют услуги цифровой связи **ADSL** (Asymmetric Digital Subscriber Line — асимметричная цифровая абонентская линия) с входящим потоком на скорости 1,5 Мбит/с и исходящим потоком в 384 кбит/с.

Таблица 10.7. Основные вызовы стандарта POSIX для управления терминалом

Вызов	Описание
s=cfsetospeed(&termios, speed)	Задать исходящую скорость
s=cfsetispeed(&termios, speed)	Задать входящую скорость
s=cfgetospeed(&termios, speed)	Получить исходящую скорость
s=cfgtetispeed(&termios, speed)	Получить входящую скорость
s=tcsetattr(fd, opt, &termios)	Задать атрибуты
s=tcgetattr(fd, &termios)	Получить атрибуты

Последние два системных вызова в списке предназначены для задания и считывания всех специальных символов, используемых для удаления символов и строк, прерывания процессов и т. д. Помимо этого они позволяют разрешить и запретить вывод эха, осуществлять управление потоком, а также выполнять другие связанные с символьным вводом-выводом функции. Также существуют дополнительные функциональные вызовы ввода-вывода, но они в некотором роде специализированные, поэтому мы не станем рассматривать их в дальнейшем. Следует также отметить, что системный вызов `ioctl` по сию пору существует во многих системах UNIX.

Реализация ввода-вывода в системе UNIX

Ввод-вывод в операционной системе UNIX реализуется набором драйверов устройств, по одному для каждого типа устройств. Функция драйвера заключается в изолировании остальной части системы от индивидуальных отличительных особенностей аппаратного обеспечения. При помощи стандартных интерфейсов между драйверами и остальной операционной системой большая часть системы ввода-вывода может быть помещена в машинно-независимую часть ядра.

Когда пользователь получает доступ к специальному файлу, файловая система определяет номера старшего и младшего устройств, а также выясняет, является ли файл блочным специальным файлом или символьным специальным файлом. Номер старшего устройства используется в качестве индекса для одного из двух внутренних массивов структур — *bdevsw* для блочных специальных файлов или *cdevsw* для символьных специальных файлов. Найденная таким образом структура содержит указатели на процедуры открытия устройства, чтения из устройства, записи на устройство и т. д. Номер младшего устройства передается в виде параметра. Добавление нового типа устройства к системе UNIX означает добавление нового элемента к одной из этих таблиц, а также предоставление соответствующих процедур выполнения различных операций с устройством.

Наиболее важные поля массива *cdevsw* для типичной системы могут выглядеть, как показано в табл. 10.8. Каждый ряд таблицы относится к одному устройству ввода-вывода (то есть одному драйверу). Колонки соответствуют функциям, которые должны поддерживаться всеми драйверами. Существуют также некоторые другие функции. Когда выполняется операция с символьным специальным файлом, система обращается к массиву *cdevsw*, выбирая соответствующий ряд (структуру), затем обращается к функции в соответствующей записи структуры (колонка таблицы), чтобы выполнить требуемое действие. Таким образом, каждый ряд таблицы содержит указатели на функции, содержащиеся в одном драйвере.

Таблица 10.8. Некоторые из полей типичной таблицы cdevsw

Устройство	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Память	null	null	mem_read	mem_write	null	...
Клавиатура	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Принтер	lp_open	lp_close	error	lp_write	lp_ioctl	...

Каждый драйвер разделен на несколько частей. Верхняя часть драйвера работает в режиме вызывающего процесса и служит интерфейсом с остальной системой UNIX. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается обращаться к процедурам ядра для выделения памяти, управления таймером, управления DMA и т. д. Набор функций ядра, которые они могут вызывать, определен в документе, называемом **Интерфейс Драйвер-Ядро**. Создание драйверов для системы UNIX подробно описано в [105].

Система ввода-вывода разделена на два основных компонента: обработку блочных специальных файлов и обработку символьных специальных файлов. Мы по очереди рассмотрим эти компоненты.

Цель той части системы, которая занимается операциями ввода-вывода с блочными специальными файлами (например, дисковым вводом-выводом), заключается в минимизации количества операций переноса данных. Для достижения данной цели в системах UNIX между дисковыми драйверами и файловой системой помещается **буферный кэш** (рис. 10.12). Буферный кэш представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Когда файловой системе требуется блок диска (например, блок i-узла, каталога или данных), сначала проверяется буферный кэш. Если нужный блок есть в кэше, он получается оттуда, при этом обращения к диску удается избежать. Буферный кэш значительно улучшает производительность системы.

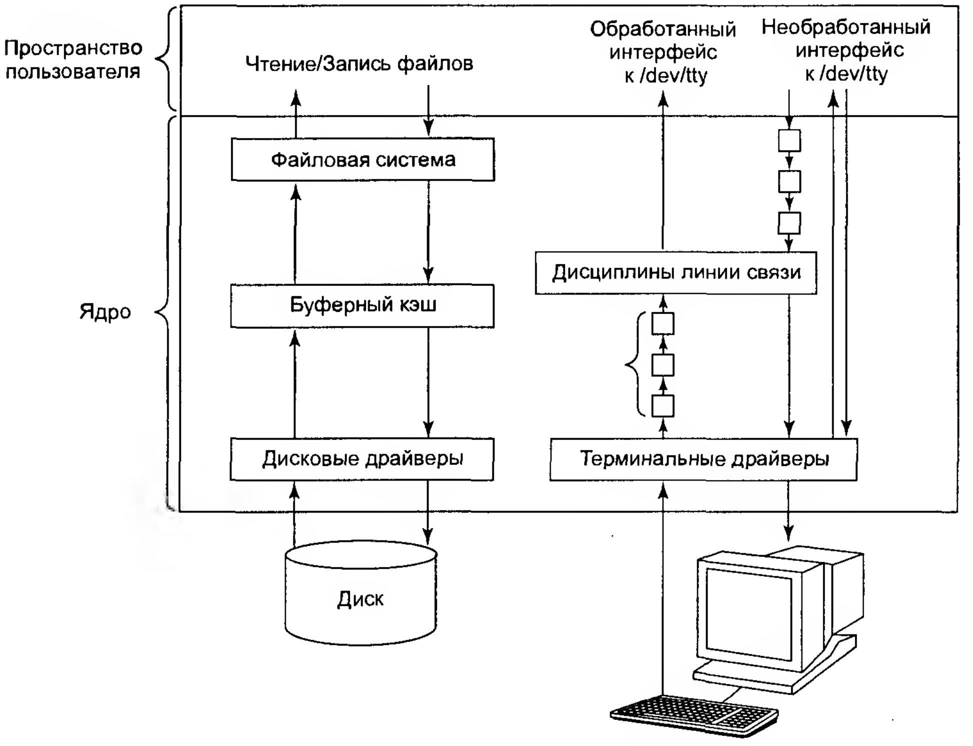


Рис. 10.12. Система ввода-вывода BSD UNIX

Если же блока нет в буферном кэше, он считывается с диска в кэш, а оттуда копируется туда, куда нужно. Поскольку в буферном кэше есть место только для фиксированного количества блоков, требуется некий алгоритм управления кэшем. Обычно блоки в кэше организуются в связный список. При каждом обращении к блоку он перемещается в начало списка. Если в кэше не хватает места для нового блока, то из него удаляется самый старый блок, находящийся в конце списка.

Буферный кэш поддерживает не только операцию чтения с диска, но также и запись на диск. Когда программа пишет блок, этот блок не попадает напрямую на диск, а отправляется в кэш. Только когда кэш наполняется, модифицированные блоки кэша сохраняются на диске. Чтобы модифицированные блоки не хранились в кэше слишком долго, принудительная выгрузка на диск «грязных» блоков производится каждые 30 с.

В течение десятилетий драйверы устройств системы UNIX статически компоновались вместе с ядром, так что все они постоянно находились в памяти при каждой загрузке системы. Такая схема хорошо работала в условиях мало меняющихся конфигураций факультетских мини-компьютеров, а затем сложных рабочих станций, то есть в тех условиях, в которых росла и развивалась операционная система UNIX. Как правило, компьютерный центр формировал ядро операционной системы, содержащее драйверы для всех необходимых в данной конфигурации устройств ввода-вывода, которыми потом и пользовался. Если на следующий год покупался новый диск, компьютерный центр перекомпоновывал ядро, что было не так уж и сложно.

Все изменилось с появлением системы Linux, ориентированной в первую очередь на поддержку персональных компьютеров. Количество всевозможных устройств ввода-вывода на персональных компьютерах на порядок больше, чем у мини-компьютеров. Кроме того, хотя у пользователей системы Linux есть (или они легко могут его получить) полный набор исходных текстов операционной системы, подавляющее большинство пользователей будет испытывать существенные трудности с добавлением нового драйвера, обновлением файлов *cdevsw* или *bdevsw*, компоновкой ядра и установкой его как загружаемой системы (не говоря уже о том, что придется делать, если построенное новое ядро откажется загружаться).

В операционной системе Linux подобные проблемы были решены при помощи концепции **подгружаемых модулей**. Это куски кода, которые могут быть загружены в ядро во время работы операционной системы. Как правило, это драйверы символьных или блочных устройств, но подгружаемым модулем также могут быть целая файловая система, сетевые протоколы, программы для отслеживания производительности системы и т. д.

При загрузке модуля должно выполняться несколько определенных действий. Во-первых, модуль должен быть на лету перенастроен на новые адреса. Во-вторых, система должна проверить, доступны ли ресурсы, необходимые драйверу (например, определенные уровни запроса прерывания), и если они доступны, то пометить их как используемые. В-третьих, должны быть настроены все необходимые векторы прерываний. В-четвертых, для поддержки нового типа старшего устройства следует обновить таблицу переключения драйверов. Наконец, драйверу позволяется выполнить любую специфическую для данного устройства процедуру инициализации. Когда все эти этапы выполнены, драйвер является полностью установленным, как и драйвер, установленный статически. Некоторые современные системы UNIX также поддерживают подгружаемые модули.

Потоки данных

Так как символьные специальные файлы имеют дело с символьными потоками, а не перемещают блоки данных между памятью и диском, они не пользуются буферным кэшем. Вместо этого в первых версиях системы UNIX каждый драйвер

символьного устройства выполнял всю работу, требуемую для данного устройства. Однако с течением времени стало ясно, что многие драйверы, например программы буферизации, управления потоком и сетевые протоколы, дублировали процедуры друг друга. Поэтому для структурирования драйверов символьных устройств и придания им модульности было разработано два решения. Мы кратко рассмотрим их по очереди.

Решение, реализованное в системе BSD, основано на структурах данных, присутствующих в классических системах UNIX, называемых **С-списками** (они показаны в виде квадратиков на рис. 10.12). Каждый С-список представляет собой блок размером до 64 символов плюс счетчик и указатель на следующий блок. Символы, поступающие с терминала или любого другого символьного устройства, буферизируются в цепочках таких блоков.

Когда пользовательский процесс считывает данные из */dev/tty* (то есть из стандартного входного потока), символы не передаются процессу напрямую из С-списков. Вместо этого они пропускаются через процедуру, расположенную в ядре и называющуюся **дисциплиной линии связи**. Дисциплина линии связи работает как фильтр, принимая необработанный поток символов от драйвера терминала, обрабатывая его и формируя то, что называется **обработанным символьным потоком**. В обработанном потоке выполняются операции локального строкового редактирования (например, удаляются отмененные пользователем символы и строки), символы возврата каретки заменяются символами переноса строки, а также выполняются другие специальные операции обработки. Обработанный поток передается процессу. Однако если процесс желает воспринимать каждый символ, введенный пользователем, он может принимать необработанный поток, минуя дисциплину линии связи.

Вывод работает аналогично, заменяя табуляторы пробелами, добавляя перед символами переноса строки символы возврата каретки, добавляя для медленных механических терминалов символы-заполнители, за которыми следует символ возврата каретки и т. д. Как и входной поток, выходной символьный поток может быть пропущен через дисциплину линии связи (обработанный режим) или миновать ее (необработанный режим). Необработанный режим особенно полезен при отправке двоичных данных на другие компьютеры по линии последовательной передачи или для графических интерфейсов пользователя. Здесь не требуется никакого преобразования.

Решение, реализованное в системе System V под называнием **потоков данных**, было разработано Деннисом Ритчи. Это общий подход (рис. 10.13). (В System V также есть буферный кэш для блочных специальных файлов, но поскольку он, по сути, не отличается от схемы, применяемой в BSD, кэш не показан здесь.) Потоки данных основаны на возможности динамически соединять процесс пользователя с драйвером, а также динамически, во время исполнения, вставлять модули обработки в поток данных. В некотором смысле поток представляет собой работающий в ядре аналог каналов в пространстве пользователя.

У потока данных всегда есть голова потока у вершины и соединение с драйвером у основания. В поток может быть вставлено столько модулей, сколько необходимо. Обработка может происходить в обоих направлениях, так что каждому модулю может понадобиться одна секция для чтения (из драйвера) и одна секция для записи (в драйвер). Когда процесс пользователя пишет данные в поток, программа в голове потока интерпретирует системный вызов и запаковывает данные

в буферы потока, передаваемые от модуля к модулю вниз, при этом каждый модуль выполняет соответствующие преобразования. У каждого модуля есть очередь чтения и очередь записи, так что буферы обрабатываются в правильном порядке. У модулей есть строго определенные интерфейсы, определяемые инфраструктурой потока, что позволяет объединять вместе несвязанные модули.

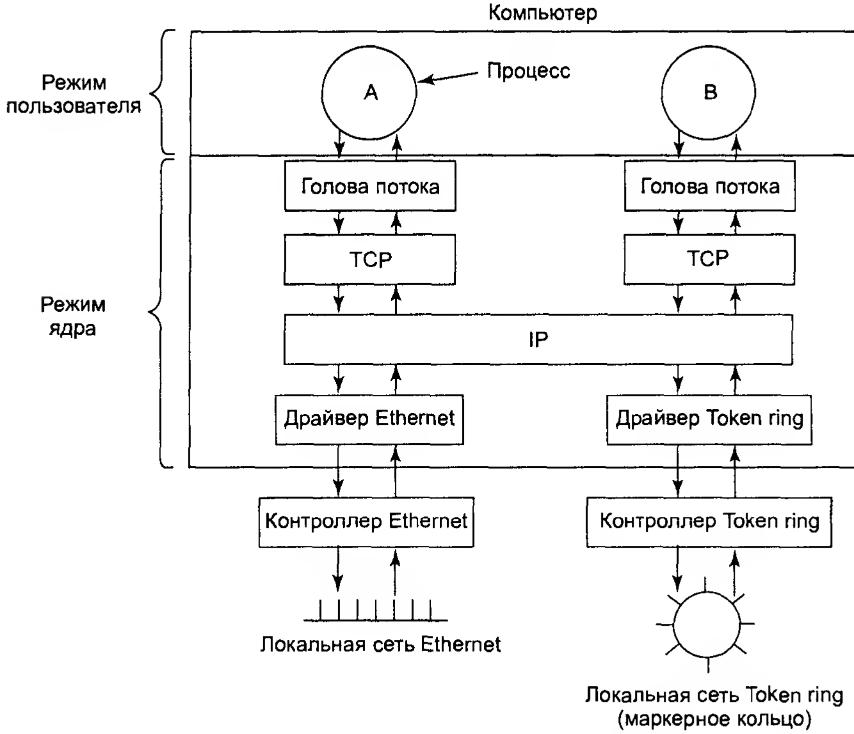


Рис. 10.13. Пример потоков в System V

В примере на рис. 10.13 показано, как применяются потоки при использовании Интернет-протокола TCP с двумя локальными сетями различного типа: Ethernet и Token ring. Данный пример иллюстрирует еще одно важное свойство потоков данных — мультиплексирование. Мультиплексный модуль может взять один поток и расщепить его на несколько потоков или, наоборот, объединить несколько потоков в единый поток. Модуль IP в данном примере выполняет обе функции.

Файловая система UNIX

Прежде всего пользователь видит в любой операционной системе, включая систему UNIX, файловую систему. В следующих разделах мы рассмотрим основные идеи файловой системы UNIX, системные вызовы и детали реализации файловой системы. Некоторые принципы файловой системы UNIX были взяты у операционной системы MULTICS, в то время как многие другие позаимствованы в MS-DOS,

Windows и других системах, хотя имеются и уникальные для UNIX идеи. Устройство файловой системы UNIX особенно интересно тем, что оно отчетливо иллюстрирует принцип разумной достаточности и красоту простых решений. При минимальном механизме и сильно ограниченном количестве системных вызовов операционная система UNIX тем не менее предоставляет мощную и элегантную файловую систему.

Основные понятия

Файл в системе UNIX — это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается принципиального различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими файлами. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Изначально размер имен файлов был ограничен 14 символами, но в системе Berkeley UNIX этот предел был расширен до 255 символов, что впоследствии было принято в System V, а также в большинстве других версий. В именах файлов разрешается использовать все ASCII-символы, кроме символа NUL, поэтому допустимы даже имена файлов, состоящие, например, из трех символов возврата каретки (хотя такое имя и не слишком удобно в использовании).

По соглашению многие программы ожидают, что имена файлов должны состоять из основного имени и расширения, отделяемого от основного имени файла точкой (которая в системе UNIX также считается символом). Так, *prog.c* — это, как правило, программа на языке C, *prog.f90* — обычно программа на языке FORTRAN 90, а *prog.o* — чаще всего объектный файл (выходные данные компилятора). Эти соглашения никак не регулируются операционной системой, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения могут иметь произвольную длину, кроме того, файлы могут иметь по нескольку расширений, как, например, *prog.java.Z*, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства использования файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и до определенного предела с ними можно работать как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и, как правило, содержит несколько подкаталогов. Символ / также используется для разделения имен каталогов, поэтому имя */usr/ast/x* означает файл *x*, расположенный в каталоге *ast*, который, в свою очередь, находится в каталоге *usr*.

Некоторые основные каталоги у вершины дерева показаны в табл. 10.9.

Таблица 10.9. Некоторые важные каталоги, существующие в большинстве систем UNIX

Каталог	Содержание
bin	Двоичные (исполняемые) программы
dev	Специальные файлы для устройств ввода-вывода
etc	Разные системные файлы
lib	Библиотеки
usr	Каталоги пользователей

Существует два способа задания имени файла в системе UNIX, как в оболочке, так и при открытии файла из программы. Первый способ заключается в использовании **абсолютного пути**, указывающего, как найти файл от корневого каталога. Пример абсолютного пути: */usr/ast/books/mos2/chap-10*. Он сообщает системе, что в корневом каталоге следует найти каталог *usr*, затем в нем найти каталог *ast*, который содержит каталог *books*, в котором содержится каталог *mos2*, а в нем, наконец, расположен файл *chap-10*.

Абсолютные имена путей часто бывают длинными и неудобными. По этой причине операционная система UNIX позволяет пользователям и процессам обозначить каталог, в котором они работают в данный момент, как **рабочий каталог** (также называемый **текущим каталогом**). Имена путей также могут указываться относительно рабочего каталога. Путь файла, заданный относительно рабочего каталога, называется **относительным путем**. Например, если каталог */usr/ast/books/mos2* является рабочим каталогом, тогда команда оболочки

```
cp chap-10 backup1
```

возымеет тот же самый эффект, что и более длинная команда

```
cp /usr/ast/books/mos2/chap-10 /usr/ast/books/mos2/backup1
```

Пользователям часто бывает необходимо обратиться к файлам, принадлежащим другим пользователям, или к своим файлам, расположенным в другом месте дерева файлов. Например, если два пользователя совместно используют один файл, он будет находиться в каталоге, принадлежащем одному из них, поэтому другому пользователю понадобится для обращения к этому файлу использовать абсолютное имя пути (или менять свой рабочий каталог). Если абсолютный путь достаточно длинен, то необходимость вводить его каждый раз может весьма сильно раздражать. В системе UNIX эта проблема решается при помощи так называемых **связей**, представляющих собой записи каталога, указывающие на другие файлы.

В качестве примера рассмотрим ситуацию на рис. 10.14, *а*. Фред и Лиза вместе работают над одним проектом, и каждому из них нужен доступ к файлам другого. Если рабочий каталог Фреда */usr/fred*, он может обращаться к файлу *x* в каталоге Лизы как */usr/lisa/x*. Однако Фред может также создать новую запись в своем каталоге (рис. 10.14, *б*), после чего он сможет обращаться к этому файлу просто как к *x*.

В только что обсуждавшемся примере мы предположили, что до создания связи единственный способ, которым Фред мог обратиться к файлу *x* Лизы, заключался в использовании абсолютного пути. В действительности это не совсем так. При создании каталога в нем автоматически создаются две записи, «.» и «..». Первая запись обозначает сам каталог. Последняя является ссылкой на родительский каталог, то есть каталог, в котором данный каталог числится как запись. Таким образом, из каталога */usr/fred* к файлу *x* Лизы можно обратиться еще и по пути *../lisa/x*.

Кроме обычных файлов, системой UNIX также поддерживаются символьные специальные файлы и блочные специальные файлы. Символьные специальные файлы используются для моделирования последовательных устройств ввода-вывода, таких как клавиатуры и принтеры. Если процесс откроет файл */dev/tty* и прочитает из него, он получит символы, введенные с клавиатуры. Если открыть файл */dev/lp* и записать в него данные, то эти данные будут распечатаны на принтере.

Блочные специальные файлы, часто с такими именами, как */dev/hd1*, могут использоваться для чтения и записи необработанных дисковых разделов, минуя файловую систему. Таким образом, поиск байта номер *k*, за которым последует чтение, приведет к чтению *k*-го байта на соответствующем дисковом разделе, игнорируя i-узел и файловую структуру. Необработанные блочные устройства используются для страничной подкачки и свопинга программами установки файловой системы (например, *mkfs*) и программами, исправляющими ломаные файловые системы (например, *fsck*).

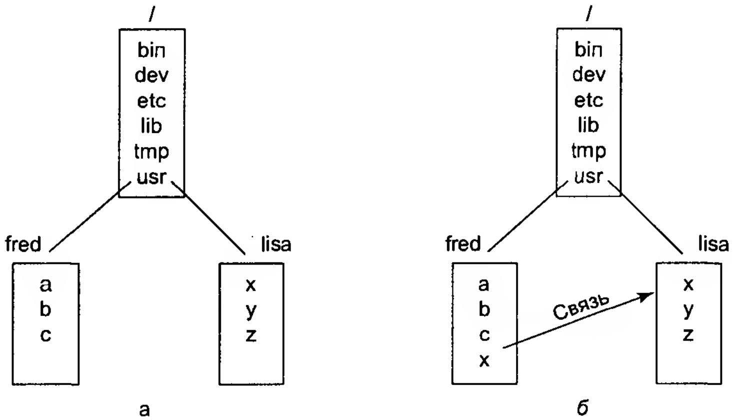


Рис. 10.14. До создания связи (*а*); после создания связи (*б*)

На многих компьютерах установлено по два и более дисков. Например, на мэйнфреймах в банках часто бывает необходимо подключать по 100 и более дисков к одной машине, чтобы хранить большие базы данных. Даже у персональных компьютеров есть по меньшей мере два диска — жесткий диск и дисковод для гибких дисков. При наличии у компьютера нескольких дисков возникает вопрос, как ими управлять.

Одно из решений заключается в том, чтобы установить самостоятельную файловую систему на каждый отдельный диск и управлять ими как отдельными файловыми системами. Рассмотрим, например, ситуацию, изображенную на рис. 10.15, *а*. Здесь показан жесткий диск, который мы будем называть *C:*, а также гибкий диск, который мы будем называть *A:*. У каждого есть собственный корневой каталог и файлы. При таком решении пользователь должен помимо каталогов указывать также и устройство, если оно отличается от используемого по умолчанию. Например, чтобы скопировать файл *x* в каталог *d* (предполагая, что по умолчанию выбирается диск *C:*), следует ввести команду

```
cp A:/x /a/d/x
```

Такой подход применяется в операционных системах MS-DOS, Windows 98 и VMS.

Решение, применяемое в операционной системе UNIX, заключается в том, чтобы позволить монтировать один диск в дерево файлов другого диска. В нашем примере мы можем смонтировать дискету в каталог */b*, получая в результате

файловую систему, показанную на рис. 10.15, *б*. Теперь пользователь видит единое дерево файлов и уже не должен думать о том, какой файл на каком устройстве хранится. В результате приведенная выше команда примет вид

```
cp /b/x /a/d/x
```

то есть все будет выглядеть так, как если бы файл копировался из одного каталога жесткого диска в другой каталог того же диска.

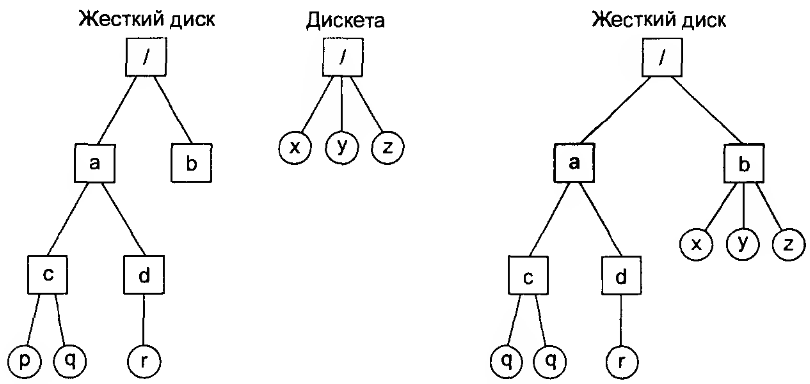


Рис. 10.15. Раздельные файловые системы (*а*); после монтирования (*б*)

Другое интересное свойство файловой системы UNIX представляет собой **блокировка**. В некоторых приложениях два и более процессов могут одновременно использовать один и тот же файл, что может привести к конфликту. Одно из решений данной проблемы заключается в том, чтобы создать в приложении критические области. Однако если эти процессы принадлежат независимым пользователям, даже не знакомым друг с другом, такой способ координации действий, как правило, очень неудобен.

Рассмотрим, например, базу данных, состоящую из многих файлов в одном или нескольких каталогах, доступ к которым могут получить никак не связанные между собой пользователи. С каждым каталогом или файлом можно связать семафор и достичь взаимного исключения, заставляя процессы выполнять операцию down на соответствующем семафоре, прежде чем читать или писать определенные данные. Недостаток этого решения заключается в том, что при этом недоступным становится весь каталог или файл, даже если процессам нужна всего одна запись.

По этой причине стандартом POSIX предоставляется гибкий и детальный механизм, позволяющий процессам за одну неделимую операцию блокировать даже единственный байт файла или целый файл, по желанию. Механизм блокировки требует от вызывающего его процесса указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, система создает запись в таблице, в которой указывается, что определенные байты файла заблокированы.

Стандартом определены два типа блокировки: **блокировка с монополизацией** и **блокировка без монополизации**. Если часть файла уже содержит блокировку без монополизации, то повторная установка блокировки без монополизации на это место файла разрешается, но попытка установки блокировку с монополизацией

будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята монопольная блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в данной области был доступен.

При установке блокировки процесс должен указать, хочет ли он сразу получить управление или будет ждать, пока не будет установлена блокировка. Если процесс выбрал вызов с ожиданием, то он блокируется до тех пор, пока с запрашиваемой области файла не будет снята блокировка, установленная другим процессом, после чего процесс активизируется, и ему сообщается, что блокировка установлена. Если процесс решил воспользоваться системным вызовом без ожидания, он немедленно получает ответ об успехе или неудаче операции.

Заблокированные области могут перекрываться. На рис. 10.16, *а* показано, что процесс *A* установил блокировку без монополизации на байты с 4 по 7 в некотором файле. Затем процесс *B* устанавливает блокировку без монополизации на байты с 6 по 9. Наконец, процесс *C* устанавливает блокировку без монополизации на байты со 2 по 11. Пока это блокировки без монополизации, они могут устанавливаться одновременно. Теперь посмотрим, что произойдет, если какой-либо процесс попытается получить блокировку с монополизацией к байту 9, используя системный вызов с ожиданием, когда блокировка данного участка файла сразу невозможна (рис. 10.16, *в*). Две предыдущие блокировки перекрываются с этой блокировкой. Поэтому обращающийся с новым запросом процесс будет заблокирован и останется заблокированным, пока оба процесса *B* и *C* не снимут свою блокировку.

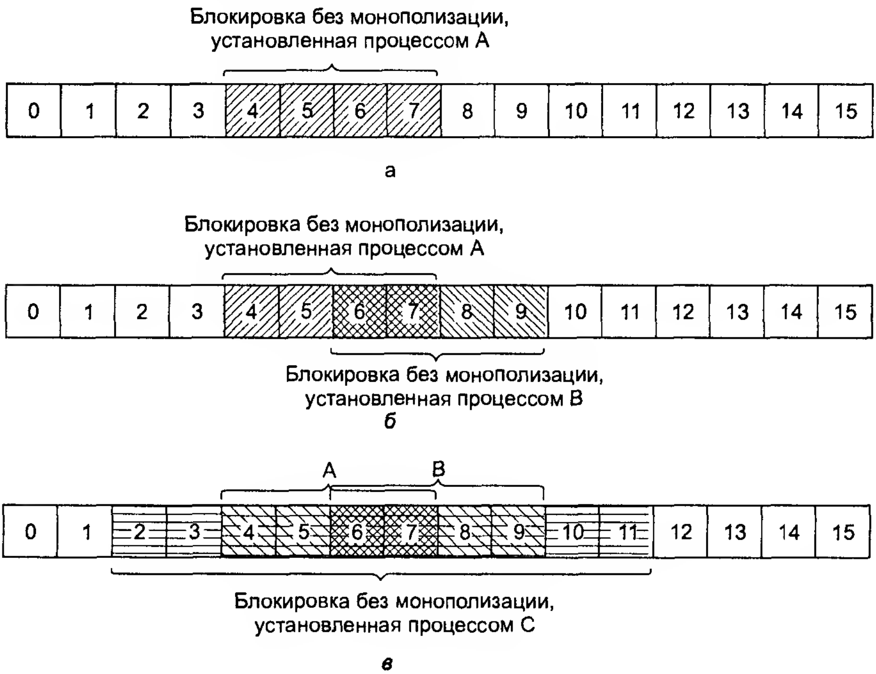


Рис. 10.16. Файл с одной блокировкой (*а*); добавление второй блокировки (*б*); третья блокировка (*в*)

Вызовы файловой системы в UNIX

Многие системные вызовы относятся к файлам и файловой системе. Сначала мы рассмотрим системные вызовы, работающие с отдельными файлами. Затем мы изучим те системные вызовы, которые оперируют каталогами или всей файловой системой в целом. Для создания нового файла можно использовать системный вызов creat. (Когда Кена Томпсона однажды спросили, что бы он поменял, если бы у него была возможность во второй раз разработать операционную систему UNIX, он ответил, что на этот раз вместо creat он назвал бы этот системный вызов create.) В качестве параметров этому системному вызову следует задать имя файла и режим защиты. Так, команда

```
fd = creat("abc", mode);
```

создает файл *abc* с режимом защиты, указанном в переменной (или константе) *mode*. Биты *mode* определяют круг пользователей, которые могут получить доступ к файлу, а также уровень предоставляемого им доступа. Вопросы защиты файлов и доступа к ним будут рассмотрены ниже.

Системный вызов creat не только создает новый файл, но также и открывает его для записи. Чтобы последующие системные вызовы могли получить доступ к файлу, успешный системный вызов creat возвращает небольшое неотрицательное целое число, называемое **дескриптором файла** (*fd* в приведенном выше примере). Если системный вызов выполняется с уже существующим файлом, длина этого файла уменьшается до 0, а все содержимое теряется.

Теперь продолжим изучение основных файловых системных вызовов, перечисленных в табл. 10.10. (В случае ошибки возвращаемое значение *s* равно –1, *fd* — дескриптор файла, *position* — смещение в файле.) Чтобы прочитать данные из существующего файла или записать данные в существующий файл, файл сначала нужно открыть с помощью системного вызова open. Этому системному вызову следует указать имя файла, а также режим, в котором он должен быть открыт: для чтения, для записи или и для того и для другого. Также можно указать различные дополнительные параметры. Как и creat, системный вызов open возвращает дескриптор файла, который может быть использован для чтения или записи журнала. Затем файл может быть закрыт при помощи системного вызова close, после чего, чтобы писать в этот файл или читать из него, его снова нужно открыть. Системные вызовы creat и open возвращают наименьший неиспользуемый в данный момент дескриптор файла.

Таблица 10.10. Некоторые системные вызовы для работы с файлами

Системный вызов	Описание
fd=creat(name, mode)	Один из способов создания нового файла
fd=open(file, how, j)	Открыть файл для чтения, записи или и того и другого
s=close(fd)	Закрыть открытый файл
n=read(fd, buffer, nbytes)	Прочитать данные из файла в буфер
n=write(fd, buffer, nbytes)	Записать данные из буфера в файл
position=lseek(fd, offset, whence)	Переместить указатель в файле
s=stat(name, &buf)	Получить информацию о состоянии файла
s=fstat(fd, &buf)	Получить информацию о состоянии файла
s=pipe(&fd[0])	Создать канал
s=fcntl(fd, cmd, ...)	Блокировка файла и другие операции

Когда программа начинает выполнение стандартным образом, файлы с дескрипторами 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках соответственно. Таким образом, фильтр, например программа *sort*, может просто читать свои входные данные из файла с дескриптором 0, а писать выходные данные в файл с дескриптором 1, не заботясь о том, где располагаются эти файлы. Работа этого механизма обеспечивается оболочкой, которая проверяет, чтобы эти дескрипторы соответствовали нужным файлам, прежде чем программа начнет свою работу.

Чаще всего программы используют системные вызовы read и write. У обоих вызовов по три параметра: дескриптор файла (указывающий, с каким из открытых файлов будет производиться операция чтения или записи), адрес буфера (сообщающий, куда положить данные или откуда их взять), а также счетчик байтов (указывающий, сколько байтов следует прочитать или записать). Вот и все. Очень простая схема. Пример типичного вызова:

```
n = read(fd, buffer, nbytes)
```

Хотя большинство программ читают и записывают файлы последовательно, некоторым программам бывает необходимо иметь доступ к произвольной части файла. С каждым открытым файлом связан указатель на текущую позицию в файле. При последовательном чтении или записи он указывает на следующий байт, который будет прочитан или записан. Например, если указатель установлен на 4096-й байт, то после успешного чтения 1024 байт из этого файла при помощи системного вызова read он будет указывать на 5120-й байт. Указатель в файле можно переместить с помощью системного вызова lseek, что позволяет при следующих обращениях к системным вызовам read и write читать данные из файла или писать их в файл в произвольной позиции в файле и даже за концом файла. Этот системный вызов назван lseek, чтобы не путать его с теперь уже устаревшим, использовавшимся ранее на 16-разрядных компьютерах системным вызовом seek.

У системного вызова lseek три параметра: первый — это дескриптор файла, второй — новая позиция в файле, а третий сообщает, указывается ли эта позиция относительно начала файла, конца файла или относительно текущей позиции. Значение, возвращаемое системным вызовом lseek, представляет собой абсолютную позицию в файле после того, как указатель был перемещен. Забавно, что системный вызов lseek (seek означает поиск, термин, также используемый для перемещения блока головок диска) никогда не вызывает перемещения блока головок диска, так как все, что он делает, — это обновление текущей позиции в файле, представляющей собой просто число в памяти.

Для каждого файла операционная система UNIX хранит такие сведения, как тип (режим) файла (обычный, каталог, специальный файл), его размер, время последнего изменения и т. д. Программы могут получить эту информацию при помощи системного вызова stat. Первый параметр представляет собой имя файла. Второй является указателем на структуру, в которую следует поместить требуемую информацию. Поля этой структуры перечислены в табл. 10.11. Системный вызов fstat представляет собой то же самое, что и системный вызов stat, с той разницей, что он работает с уже открытым файлом (имя которого может быть неизвестно), а не с путем.

Таблица 10.11. Поля структуры, возвращаемой системным вызовом stat

Устройство, на котором располагается файл
Номер i-узла (идентифицирует файл на устройстве)
Режим файла (включая информацию о защите)
Количество связей файла
Идентификатор владельца файла
Группа, к которой принадлежит файл
Размер файла в байтах
Время создания
Время последнего доступа
Время последнего изменения

Системный вызов pipe используется для создания каналов оболочки. Он создает псевдофайл для буферирования данных, которыми обмениваются компоненты канала, и возвращает дескрипторы файлов для чтения и записи буфера. В команде

```
sort <in | head -30
```

дескриптор файла 1 (стандартный вывод) в процессе, выполняющем программу *sort*, будет настроен оболочкой на запись в канал, а дескриптор файла 0 (стандартный ввод) в процессе, выполняющем программу *head*, будет настроен на чтение из канала. Программа *sort* просто читает из файла с дескриптором 0 (установлен на файл *in*) и пишет в файл с дескриптором 1 (канал), даже не зная, что оба эти файла переопределены. Если бы ввод и вывод не были перенаправлены, программа *sort* читала бы данные с клавиатуры и выводила бы их на экран (устройства по умолчанию). Подобным же образом, когда программа *head* считывает входные данные из файла с дескриптором 0, она получает данные, которые программа *sort* поместила в буфер канала, даже не зная, что канал используется. Вот пример того, как простая концепция (перенаправление потока данных) плюс простая реализация (файлы с дескрипторами 0 и 1) вместе дают мощный инструмент (соединение программ произвольным образом без необходимости их модификации).

Последний системный вызов в табл. 10.10 — это fcntl. Он используется для блокировки и разблокирования файлов, а также некоторых других специфических для файлов операций.

Рассмотрим теперь некоторые системные вызовы, относящиеся скорее к каталогам или файловой системе в целом, нежели к какому-либо конкретному файлу. Наиболее часто употребляемые системные вызовы перечислены в табл. 10.12. (В случае ошибки возвращаемое значение *s* равно –1, *dir* идентифицирует каталог, а *dirent* представляет собой запись каталога.) Каталоги создаются и удаляются при помощи системных вызовов mkdir и rmdir соответственно. Каталог может быть уничтожен, только если он пуст.

Как было показано на рис. 10.14, при создании связи с файлом создается новая запись в каталоге, указывающая на существующий файл. Связь создается при помощи системного вызова link. В параметрах этого системного вызова указываются исходное и новое имя. Записи в каталоге удаляются системным вызовом unlink. Когда удаляется последняя связь с файлом, файл также автоматически удаляется. Если для файла не было создано ни одной связи, то при первом же обращении к системному вызову unlink файл будет удален.

Таблица 10.12. Некоторые системные вызовы, имеющие отношение к работе с каталогом

Системный вызов	Описание
s=mkdir(path, mode)	Создать новый каталог
s=rmdir(path)	Удалить каталог
s=link(oldpath, newpath)	Создать связь с существующим файлом
s=unlink(path)	Удалить связь
s=chdir(path)	Изменить рабочий каталог
dir=opendir(path)	Открыть каталог для чтения
s=closedir(dir)	Закрыть каталог
dirent=readdir(dir)	Прочитать одну запись каталога
rewinddir(dir)	Установить указатель в каталоге на первую запись

Рабочий каталог можно изменить при помощи системного вызова `chdir`. После выполнения этого системного вызова будут по-другому интерпретироваться относительные имена путей.

Последние четыре системных вызова в табл. 10.12 предназначены для чтения каталогов. Каталоги могут открываться, закрываться и читаться аналогично обычным файлам. Каждое обращение к системному вызову `readdir` возвращает ровно одну запись каталога фиксированного формата. Пользователям запрещено писать в каталоги (это делается, чтобы пользователи случайно не нарушили целостности системы). Файлы могут добавляться к каталогу при помощи системных вызовов `creat` и `link`, а удаляться с помощью системного вызова `unlink`. В операционной системе UNIX нет способа обращаться к файлу по расположению его описателя в каталоге, но есть системный вызов `rewinddir`, позволяющий начать читать открытый каталог с начала.

Реализация файловой системы UNIX

В данном разделе будет описана реализация традиционной файловой системы UNIX. Затем мы обсудим усовершенствования, реализованные в версии Berkeley. Также используются и другие файловые системы. Все системы UNIX могут поддерживать несколько дисковых разделов, каждый со своей файловой системой.

В классической системе UNIX раздел диска содержит файловую систему, расположение которой изображено на рис. 10.17. Блок 0 не используется системой и часто содержит программу загрузки компьютера. Блок 1 представляет собой **суперблок**. В нем хранится критическая информация о размещении файловой системы, включая количество i-узлов, количество дисковых блоков, а также начало списка свободных блоков диска (обычно несколько сот записей). При уничтожении суперблока файловая система окажется нечитаемой.

Следом за суперблоком располагаются **i-узлы** (i-nodes, сокращение от index-nodes — индекс-узлы). Они нумеруются от 1 до некоторого максимального числа. Каждый i-узел имеет 64 байт в длину и описывает ровно один файл. i-узел содержит учетную информацию (включая всю информацию, возвращаемую системным вызовом `stat`, который ее просто берет в i-узле), а также достаточное количество информации, чтобы найти все блоки файла на диске.

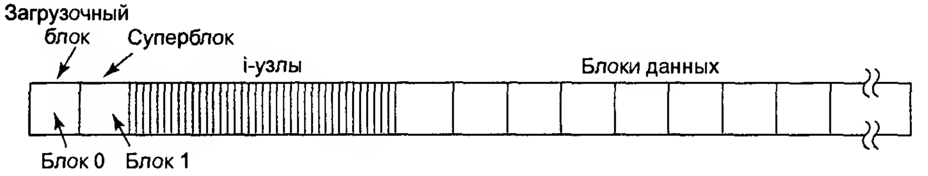


Рис. 10.17. Расположение классической файловой системы UNIX на диске

Следом за i-узлами располагаются блоки с данными. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, блоки файла не обязаны располагаться на диске подряд. В действительности блоки большого файла, как правило, оказываются разбросанными по всему диску. Именно эту проблему должны были решить усовершенствования версии Berkeley.

Каталог в традиционной файловой системе (то есть V7) представляет собой несортированный набор 16-байтовых записей. Каждая запись состоит из 14-байтного имени файла и номера i-узла (см. рис. 6.33). Чтобы открыть файл в рабочем каталоге, система просто считывает каталог, сравнивая имя искомого файла с каждой записью, пока не найдет нужную запись или пока не закончится каталог.

Если искомый файл присутствует в каталоге, система извлекает его i-узел и использует его в качестве индекса в таблице i-узлов (на диске), чтобы найти соответствующий i-узел и считать его в память. Этот i-узел помещается в **таблицу i-узлов**, структуру данных в ядре, содержащую все i-узлы открытых в данный момент файлов и каталогов. Формат i-узлов варьируется от одной версии UNIX к другой. Как минимум i-узел должен содержать все поля, возвращаемые системным вызовом stat (см. табл. 10.11). В табл. 10.13 показана структура i-узла, используемая в AT&T версиях системы UNIX от Version 7 до System V.

Таблица 10.13. Структура i-узла в System V

Поле	Байты	Описание
Mode	2	Тип файла, биты защиты, биты setuid и setgid
Nlinks	2	Количество каталоговых записей, указывающих на этот i-узел
Uid	2	UID владельца файла
Gid	2	GID владельца файла
Size	4	Размер файла в байтах
Addr	39	Адрес первых 10 дисковых блоков файла и 3 косвенных блоков
Gen	1	Номер «поколения» (увеличивается на единицу при каждом использовании i-узла)
Atime	4	Время последнего доступа к файлу
Mtime	4	Время последнего изменения файла
Ctime	4	Время последнего изменения i-узла (не считая других раз)

Поиск файла по абсолютному пути, например */usr/ast/file*, немного сложнее. Сначала система находит корневой каталог, как правило, использующий i-узел с номером 2 (i-узел 1 обычно резервируется для хранения дефектных блоков). Затем он ищет в корневом каталоге строку «usr», чтобы получить номер i-узла каталога */usr*. Затем считывается этот i-узел, и из него извлекаются номера блоков, в которых

располагается каталог */usr*. После этого считывается каталог */usr*, в котором ищется строка «ast». Когда нужная запись найдена, из нее извлекается номер i-узла для каталога */usr/ast* и т. д. Таким образом, использование относительного имени файла не только удобнее для пользователя, но также представляет существенно меньшее количество работы для файловой системы.

Рассмотрим теперь, как система считывает файл. Вспомним, что типичное обращение к библиотечной процедуре для запуска системного вызова read выглядит следующим образом:

```
n = read(fd. buffer. nbytes):
```

Когда ядро получает управление, ему подаются только эти три параметра. Все остальные необходимые данные оно может получить из внутренних таблиц, относящихся к пользователю. Одной из таких таблиц является массив дескрипторов файла. Он проиндексирован по номерам дескрипторов файла и содержит по одной записи для каждого открытого файла (до некоторого максимума, как правило, около 20 файлов).

По дескриптору файла файловая система должна найти i-узел соответствующего файла. Рассмотрим одно из возможных решений: просто поместим в таблицу дескрипторов файла указатель на i-узел. Несмотря на простоту, данный метод, увы, не работает. Проблема заключается в следующем. С каждым дескриптором файла должен быть связан указатель в файле, определяющий байт в файле, который будет считан или записан при следующем обращении к файлу. Где следует хранить этот указатель? Один вариант состоит в помещении его в таблице i-узлов. Однако такой подход не сможет работать, если несколько не связанных друг с другом процессов одновременно откроют один и тот же файл, так как у каждого процесса должен быть свой собственный указатель.

Второй вариант решения заключается в помещении указателя в таблицу дескрипторов файла. При этом каждый процесс, открывающий файл, получает собственную позицию в файле. К сожалению, такая схема также не работает, но причина неудачи в данном случае не столь очевидна и имеет отношение к природе совместного использования файлов в системе UNIX. Рассмотрим сценарий оболочки *s*, состоящий из двух команд, *p*1 и *p*2, которые должны работать по очереди. Если сценарий вызывается командной строкой

```
s >x
```

то ожидается, что команда *p*1 будет писать свои выходные данные в файл *x*, а команда *p*2 будет также писать свои выходные данные в файл *x*, начиная с того места, на котором остановилась команда *p*1.

Когда оболочка запустит процесс *p*1, файл *x* будет изначально пустым, поэтому команда *p*1 просто начнет запись в файл в позиции 0. Однако когда она закончит свою работу, требуется некий механизм передачи указателя в файле *x* от процесса *p*1 процессу *p*2. Если же позицию в файле хранить просто в таблице дескрипторов файла, процесс *p*2 начнет запись в файл с позиции 0.

Способ передачи указателя от одного процесса другому показан на рис. 10.18. Трюк состоит в том, чтобы ввести в обращение новую таблицу, **таблицу открытых файлов**, между таблицей дескрипторов файлов и таблицей i-узлов, и хранить

указатели в файле, а также бит чтения/записи в ней. На рисунке родительский процесс представляет собой оболочку, а дочерний процесс сначала является процессом $p1$, а затем процессом $p2$. Когда оболочка создает процесс $p1$, его пользовательская структура (включая таблицу дескрипторов файлов) представляет собой точную копию такой же структуры оболочки, поэтому обе они содержат указатели на одну и ту же таблицу открытых файлов. Когда процесс $p1$ завершает свою работу, таблица дескрипторов файлов оболочки продолжает указывать на таблицу открытых файлов, в которой содержится позиция в файле процесса $p1$. Когда теперь оболочка создает процесс $p2$, новый дочерний процесс автоматически наследует позицию в файле. При этом ни сам новый процесс, ни оболочка даже не должны знать текущее значение этой позиции.

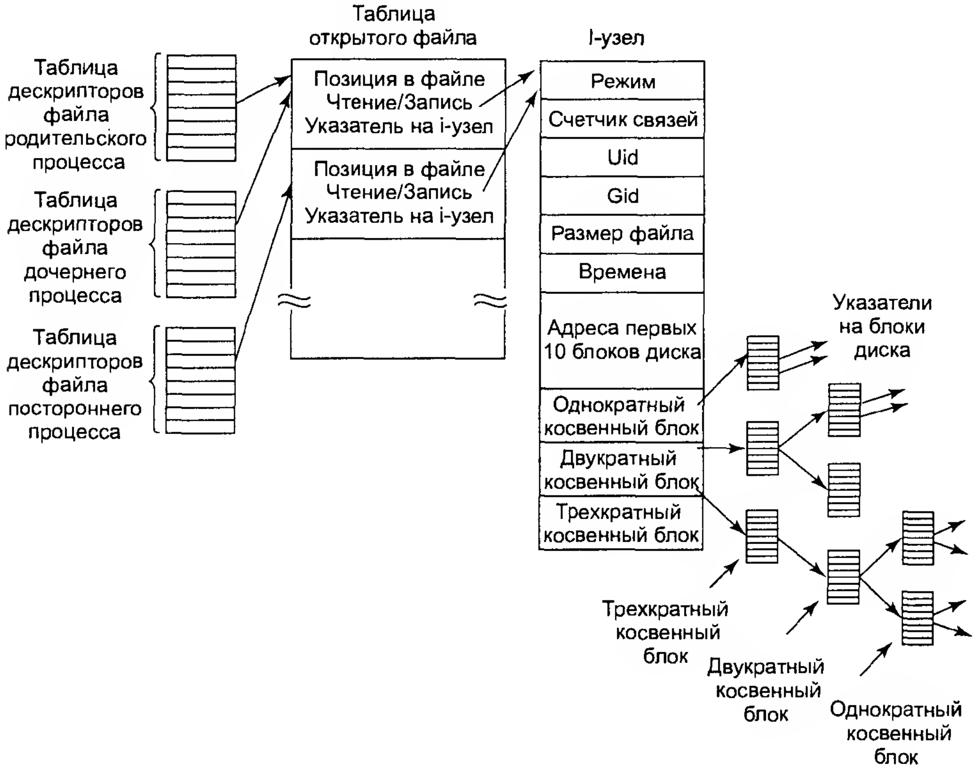


Рис. 10.18. Связь между таблицей дескрипторов файлов, таблицей открытых файлов и таблицей i-узлов

Однако если какой-нибудь посторонний процесс откроет файл, он получит свою собственную запись в таблице открытых файлов со своей позицией в файле, что как раз и нужно. Таким образом, задача таблицы открытых файлов заключается в том, чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять отдельные указатели.

Итак, мы показали, как работающие процессы получают доступ к позиции в файле и к i-узлу файла. i-узел содержит дисковые адреса первых 10 блоков файла. Если позиция в файле попадает в его первые 10 блоков, то считывается нужный блок файла, а данные копируются пользователю. Для поддержки файлов, длина которых превышает 10 блоков, в i-узле содержится дисковый адрес **одинарного косвенного блока** (см. рис. 10.18). Этот блок содержит дисковые адреса дополнительных блоков файла. Например, если размер блока составляет 1 Кбайт, а дисковый адрес занимает 4 байт, то одинарный косвенный блок может хранить до 256 дисковых адресов. Такая схема позволяет поддержать файлы размером до 266 Кбайт.

Для файлов, размер которых превосходит 266 Кбайт, используется **двойной косвенный блок**. Он содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных. Такая схема позволяет поддержать файлы размером до 10 + 2^{16} блоков (67 119 104 байт). Если и этого оказывается недостаточно, в i-узле есть место для **тройного косвенного блока**. Его указатели показывают на 256 двойных косвенных блоков.

Файловая система Berkeley Fast

Приведенное выше описание объясняет принципы работы классической файловой системы UNIX. Теперь познакомимся с усовершенствованиями этой системы, реализованными в версии Berkeley. Во-первых, были реорганизованы каталоги. Длина имен файлов была увеличена до 255 символов. Конечно, изменение структуры всех каталогов означало, что программы, продолжающие наивно читать напрямую содержимое каталогов (что было и остается допустимым) и ожидающие обнаружить в них последовательность 16-байтовых записей, более не работали. Для обеспечения совместимости двух систем в системе Berkeley были разработаны системные вызовы `opendir`, `closedir`, `readdir` и `rewinddir`, чтобы программы могли читать каталоги, не зная их внутренней структуры. Позднее длинные имена файлов и эти системные вызовы были добавлены ко всем другим версиям UNIX и к стандарту POSIX.

Структура каталогов BSD, поддерживающая имена файлов длиной до 255 символов, показана на рис. 10.19. Каждый каталог состоит из некоторого целого количества дисковых блоков, так что каталоги могут записываться на диск как единое целое. Внутри каталога записи файлов и каталогов никак не отсортированы, при этом каждая запись сразу следует за предыдущей записью. В конце каждого блока может оказаться несколько неиспользованных байтов, так как записи могут быть различного размера.

Каждая каталоговая запись на рис. 10.19 состоит из четырех полей фиксированной длины и одного поля переменной длины. Первое поле представляет собой номер i-узла, равный 19 для файла *colossal*, 42 для файла *voluminous* и 88 для каталога *bigdir*. Следом за номером i-узла идет поле, сообщающее размер всей каталоговой записи в байтах, возможно, вместе с дополнительными байтами-заполнителями в конце записи. Это поле необходимо, чтобы найти следующую запись. На рисунке это поле обозначено стрелкой, указывающей на начало следующей записи. Затем располагается поле типа файла, определяющее, является ли этот файл каталогом и т. д. Последнее поле содержит длину имени файла в байтах (8, 10 и 6 для данного

примера). Наконец, идет само имя файла, заканчивающееся нулевым байтом и дополненное до 32-битовой границы. За ним могут следовать дополнительные байты-заполнители.

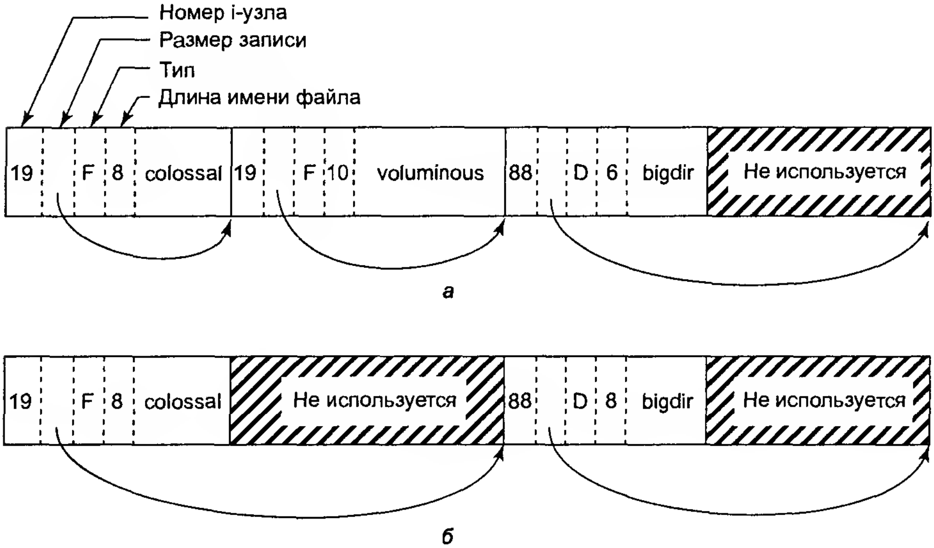


Рис. 10.19. Каталог BSD с тремя файлами (*а*); тот же каталог после удаления файла voluminous (*б*)

На рис. 10.19, *б* показан тот же самый каталог после того, как файл *voluminous* был удален. Все, что при этом делается в каталоге, — увеличивается размер записи предыдущего файла *colossal*, а байты каталоговой записи удаленного файла *voluminous* превращаются в заполнители первой записи. Впоследствии эти байты могут использоваться для записи при создании нового файла.

Поскольку поиск в каталогах производится линейно, он может занять много времени, пока не будет найдена запись у конца большого каталога. Для увеличения производительности в BSD было добавлено кэширование имен. Прежде чем искать имя в каталоге, система проверяет кэш. Если имя файла есть в кэше, то в каталоге его уже можно не искать.

Вторым существенным изменением, введенным в Berkeley, было разбиение диска на **группы цилиндров**, у каждой из которых был собственный суперблок, i-узлы и блоки данных. Идея такой организации диска заключается в том, чтобы хранить i-узел и блоки данных файла ближе друг к другу. Тогда при обращении к файлам снижается время, затрачиваемое жестким диском на перемещение блоков головок. По мере возможности блоки для файла выделяются в группе цилиндров, в которой содержится i-узел.

Третье изменение заключалось в использовании блоков не одного, а двух размеров. Для хранения больших файлов значительно эффективнее использовать небольшое количество крупных блоков, чем много маленьких блоков. С другой стороны, размер многих файлов в системе UNIX невелик, поэтому при использовании только блоков большого размера расходовалось бы слишком много диско-

вого пространства. Наличие блоков двух размеров обеспечивает эффективное чтение/запись для больших файлов и эффективное использование дискового пространства для небольших файлов. Платой за эффективность является значительная дополнительная сложность программы.

Файловая система Linux

Изначально в операционной системе Linux использовалась файловая система операционной системы MINIX. Однако в системе MINIX длина имен файлов ограничивалась 14 символами (для совместимости с UNIX Version 7), а максимальный размер файла был равен 64 Мбайт. Поэтому у разработчиков операционной системы Linux практически сразу появился интерес к усовершенствованию файловой системы. Первым шагом вперед стала файловая система Ext, в которой длина имен файлов была увеличена до 255 символов, а размер файлов — до 2 Гбайт. Однако эта система была медленнее файловой системы MINIX, поэтому исследования некоторое время продолжались. Наконец, была разработана файловая система Ext2, с длинными именами файлов, длинными файлами и высокой производительностью. Эта файловая система и стала основной файловой системой Linux. Однако операционная система Linux также поддерживает еще более десятка файловых систем, используя для этого файловую систему NFS (описанную в следующем разделе). При компоновке операционной системы Linux предлагается сделать выбор файловой системы, которая будет встроена в ядро. Другие файловые системы при необходимости могут динамически подгружаться во время исполнения в виде модулей.

Файловая система Ext2 очень похожа на файловую систему Berkeley Fast File system с небольшими изменениями. Размещение файловой системы Ext2 на жестком диске показано на рис. 10.20. Вместо того чтобы использовать группы цилиндров, что практически ничего не значит при современных дисках с виртуальной геометрией, она делит диск на группы блоков, независимо от того, где располагаются границы между цилиндрами. Каждая группа блоков начинается с суперблока, в котором хранится информация о том, сколько блоков и i-узлов находятся в данной группе, о размере группы блоков и т. д. Затем следует описатель группы, содержащий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также количестве каталогов в группе. Эта информация важна, так как файловая система Ext2 пытается распространить каталоги равномерно по всему диску.

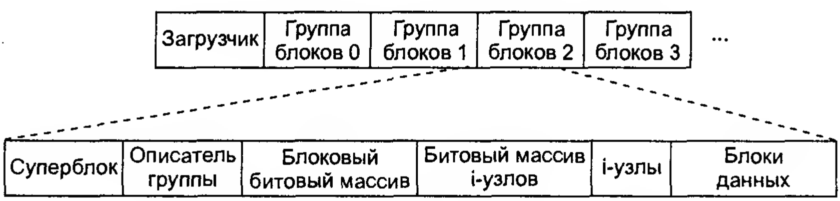


Рис. 10.20. Размещение файловой системы Ext2 на диске

В двух битовых массивах ведется учет свободных блоков и свободных i-узлов. Размер каждого битового массива равен одному блоку. При размере блоков в 1 Кбайт такая схема ограничивает размер группы блоков 8192 блоками и 8192 i-узлами. (На практике ограничение числа i-узлов никогда не встречается, так как блоки

заканчиваются раньше.) Затем располагаются сами i-узлы. Размер каждого i-узла — 128 байт, что в два раза больше размера стандартных i-узлов в UNIX (см. табл. 10.13). Дополнительные байты в i-узле используются следующим образом. Вместо 10 прямых и 3 косвенных дисковых адресов файловая система Linux позволяет 12 прямых и 3 косвенных дисковых адреса. Кроме того, длина адресов увеличена с 3 до 4 байт, и это позволяет поддерживать дисковые разделы размером более 2^{24} блоков (16 Гбайт), что уже стало проблемой для UNIX. Помимо этого зарезервированы поля для указателей на списки управления доступом, что должно обеспечить более высокую степень защиты, но это пока находится на стадии проектирования. Остаток i-узла зарезервирован на будущее. Как показывает история, неиспользуемые биты недолго остаются таковыми.

Работа файловой системы похожа на функционирование быстрой файловой системы Berkeley. Однако в отличие от BSD, в системе Linux используются дисковые блоки только одного размера, 1 Кбайт. Быстрая файловая система Berkeley использует 8-килобайтные блоки, которые затем разбиваются при необходимости на килобайтные фрагменты. Файловая система Ext2 делает примерно то же самое, но более простым способом. Как и система Berkeley, когда файл увеличивается в размерах, файловая система Ext2 пытается поместить новый блок файла в ту же группу блоков, что и остальные блоки, желательно сразу после предыдущих блоков. Кроме того, при создании нового файла в каталоге файловая система Ext2 старается выделить ему блоки в той же группе блоков, в которой располагается каталог. Новые каталоги, наоборот, равномерно распределяются по всему диску.

Другой файловой системой Linux является файловая система **/proc** (process — процесс). Идея этой файловой системы изначально была реализована в 8-й редакции операционной системы UNIX, созданной лабораторией Bell Labs, а позднее скопированной в 4.4BSD и System V. Однако в операционной системе Linux данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается подкаталог в каталоге */proc*. Имя подкаталога формируется из PID процесса в десятичном формате. Например, */proc619* — это каталог, соответствующий процессу с PID 619. В этом каталоге располагаются файлы, которые хранят информацию о процессе — его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от фактического процесса и возвращает ее в стандартном формате.

Многие расширения, реализованные в операционной системе Linux, относятся к файлам и каталогам, расположенным в каталоге */proc*. Они содержат информацию о центральном процессоре, дисковых разделах, векторах прерывания, счетчиках ядра, файловых системах, подгружаемых модулях и о многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы безопасным способом. Некоторые из этих файлов могут записываться в каталог */proc*, чтобы изменить параметры системы.

Файловая система NFS

Сети играют главную роль в операционной системе UNIX с самого начала (первая сеть в системе UNIX была построена для переноса нового ядра с машины PDP-11/70 на компьютер Interdata 8/32). В данном разделе мы рассмотрим фай-

ловую систему **NFS** (Network File System — сетевая файловая система) корпорации Sun Microsystems, использующуюся на всех современных системах UNIX (а также на некоторых не-UNIX системах) для объединения на логическом уровне файловых систем отдельных компьютеров в единое целое. Интересны три аспекта файловой системы NFS: архитектура, протокол и реализация. Мы рассмотрим их все по очереди.

Архитектура файловой системы NFS

В основе файловой системы NFS лежит представление о том, что пользоваться общей файловой системой может произвольный набор клиентов и серверов. Во многих случаях все клиенты и серверы располагаются на одной и той же локальной сети, хотя этого не требуется. Файловая система NFS может также работать в глобальной сети, если сервер находится далеко от клиента. Для простоты мы будем говорить о клиентах и серверах, как если бы они работали на различных компьютерах, хотя файловая система NFS позволяет каждой машине одновременно быть клиентом и сервером.

Каждый сервер файловой системы NFS экспортирует один или несколько ее каталогов, предоставляя доступ к ним удаленным клиентам. Как правило, доступ к каталогу предоставляется вместе со всеми его подкаталогами, то есть все дерево каталогов экспортируется как единое целое. Список экспортируемых сервером каталогов хранится в файле */etc/exports*, таким образом, эти каталоги экспортируются автоматически при загрузке сервера. Клиенты получают доступ к экспортируемым каталогам, монтируя эти каталоги. Когда клиент монтирует (удаленный) каталог, этот каталог становится частью иерархии каталогов клиента, как показано на рис. 10.21. (Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков.)

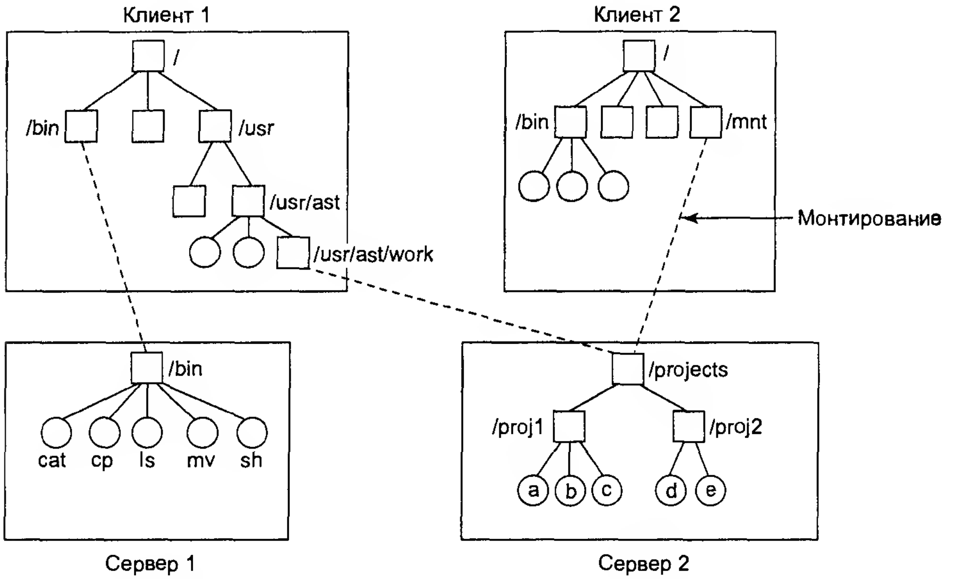


Рис. 10.21. Примеры монтирования удаленных файловых систем

В этом примере клиент 1 смонтировал каталог *bin* сервера 1 в своем собственном каталоге *bin*, поэтому он теперь может обращаться к оболочке сервера 1 как к *bin/sh*. У бездисковых рабочих станций часто есть только скелет файловой системы (в ОЗУ), а все свои файлы они получают с удаленных серверов, как в данном примере. Аналогично клиент 1 смонтировал каталог *projects* сервера 1 в своем каталоге *usr/ast/work*, поэтому он теперь может получать доступ к файлу *a* как к *usr/ast/work/proj1/a*. Как видно из этого примера, у одного и того же файла могут быть различные имена на различных клиентах, так как их каталоги могут монтироваться в различных узлах каталоговых деревьев. Выбор узла, в котором монтируется удаленный каталог, целиком зависит от клиента. Сервер не знает, где клиент монтирует его каталог.

Протоколы файловой системы NFS

Так как одна из целей файловой системы NFS заключается в поддержке разнородных систем, в которых клиенты и серверы могут работать под управлением различных операционных систем и на различном оборудовании, существенно, чтобы интерфейс между клиентами и серверами был тщательно определен. Только в этом случае можно ожидать, что новый написанный клиент будет корректно работать с существующими серверами, и наоборот.

В файловой системе NFS эта задача выполняется при помощи двух протоколов клиент-сервер. **Протокол** — это набор запросов, посылаемых клиентами серверам, и ответов серверов, посылаемых клиентам.

Первый протокол NFS управляет монтированием каталогов. Клиент может послать серверу путь к каталогу и запросить разрешение смонтировать этот каталог где-либо в своей иерархии каталогов. Данные о месте, в котором клиент намеревается смонтировать удаленный каталог, серверу не посылаются, так как серверу это безразлично. Если путь указан верно и указанный каталог был экспортирован, тогда сервер возвращает клиенту **дескриптор файла**, содержащий поля, однозначно идентифицирующие тип файловой системы, диск, i-узел каталога и информацию о правах доступа. Этот дескриптор файла используется последующими обращениями чтения и записи к файлам в монтированном каталоге или в любом из его подкаталогов.

Во время загрузки операционная система UNIX, прежде чем перейти в многопользовательский режим, запускает сценарий оболочки */etc/rc*. В этом сценарии можно разместить команды монтировки файловых систем. Таким образом, все необходимые удаленные файловые системы будут автоматически смонтированы прежде, чем будет разрешена регистрация в системе. В качестве альтернативы в большинстве версий системы UNIX также поддерживается **автомонтировка**. Это свойство позволяет ассоциировать с локальным каталогом несколько удаленных каталогов. Ни один из этих удаленных каталогов не монтируется во время загрузки операционной системы (не происходит даже контакта с сервером). Вместо этого при первом обращении к удаленному файлу (когда файл открывается) операционная система посылает каждому серверу сообщение. Побеждает ответивший первым сервер, чей каталог и монтируется.

У автомонтировки есть два принципиальных преимущества перед статической монтировкой с использованием файла */etc/rc*. Во-первых, если один из серверов,

перечисленных в файле */etc/rc*, окажется выключенным, запустить клиента будет невозможно, по крайней мере без определенных трудностей, задержки и большого количества сообщений об ошибках. Если пользователю в данный момент этот сервер не нужен, вся работа просто окажется напрасной. Во-вторых, предоставление клиенту возможности связаться с несколькими серверами параллельно позволяет значительно повысить устойчивость системы к сбоям (так как для работы достаточно всего одного работающего сервера) и улучшить показатели производительности (так как первый ответивший сервер скорее всего окажется наименее загруженным).

С другой стороны, при таком подходе неявно подразумевается, что все указанные как альтернативные файловые системы идентичны для автомонтировки. Так как файловая система NFS не предоставляет поддержки репликации файлов или каталогов, то следить за идентичностью всех файловых систем должен сам пользователь. Поэтому автомонтировка, как правило, используется для файловых систем, в которых клиенту разрешено только чтение. Такие файловые системы обычно содержат системные двоичные файлы, а также другие редко изменяемые файлы.

Второй протокол NFS предназначен для доступа к каталогам и файлам. Клиенты могут посылать серверам сообщения, содержащие команды управления каталогами и файлами, что позволяет им создавать, удалять, читать и писать файлы. Кроме того, у клиентов есть доступ к атрибутам файла, таким как режим, размер и время последнего изменения файла. Файловой системой NFS поддерживается большинство системных вызовов операционной системы UNIX, за исключением, как ни странно, системных вызовов open и close.

Пропуск системных вызовов open и close не случаен. Это сделано намеренно. Нет необходимости открывать файл, прежде чем прочитать его. Также не нужно закрывать файл после того, как данные из него прочитаны. Вместо этого, чтобы прочитать файл, клиент посылает на сервер сообщение lookup, содержащее имя файла, с запросом найти этот файл и вернуть дескриптор файла, представляющий собой структуру, идентифицирующую файл (то есть содержащую идентификатор файловой системы и номер i-узла вместе с прочей информацией). В отличие от системного вызова open, операция lookup не копирует никакой информации во внутренние системные таблицы. Системному вызову read подается на входе дескриптор файла, который предстоит прочитать, смещение в файле, а также количество байтов, которые нужно прочитать. Таким образом, каждое сообщение является самодостаточным. Преимущество такой схемы заключается в том, что серверу не нужно помнить что-либо об открытых соединениях между обращениями к нему. Поэтому если на сервере произойдет сбой с последующей перезагрузкой, не будет потеряно никакой информации об открытых файлах, так как терять просто нечего. Такие серверы называются **серверами без состояния**.

К сожалению, метод файловой системы NFS усложняет достижение точной файловой семантики системы UNIX. Например, в операционной системе UNIX файл может быть открыт и заблокирован, так что никакой другой процесс не смог получить к этому файлу доступ. Когда файл закрывается, все его блокировки снимаются. В сервере без состояния, как в файловой системе NFS, с открытыми файлами нельзя связать блокировку, так как сервер не знает, какие файлы открыты. Следовательно, в файловой системе NFS требуется отдельный специальный механизм осуществления блокировки.

Файловая система NFS использует стандартный механизм защиты UNIX с битами *rwx* для владельца, группы и всех прочих пользователей (этот вопрос упоминался в главах 1 и 6, а также подробно обсуждается ниже). Изначально каждое сообщение с запросом просто содержало идентификаторы пользователя и группы вызывающего процесса, которые сервер NFS использовал для проверки прав доступа. Сервер NFS предполагал, что клиенты не будут его обманывать. Несколько лет опыта работы с файловой системой NFS показали, что такое предположение было (как бы это помягче назвать?) наивным. Сегодня для установки надежного ключа для аутентификации клиента и сервера при каждом запросе и каждом ответе можно использовать шифрование с открытым ключом. При этом злоумышленник не сможет выдать себя за другого клиента (или другой сервер), так как ему неизвестен секретный ключ этого клиента (или сервера).

Реализация файловой системы NFS

Хотя реализация программ клиента и сервера не зависит от протоколов NFS, в большинстве систем UNIX используется трехуровневая реализация, сходная с изображенной на рис. 10.22. Верхний уровень представляет собой уровень системных вызовов. Он управляет такими системными вызовами, как open, read и close. После анализа системного вызова и проверки его параметров он вызывает второй уровень, уровень VFS (Virtual File System — виртуальная файловая система).

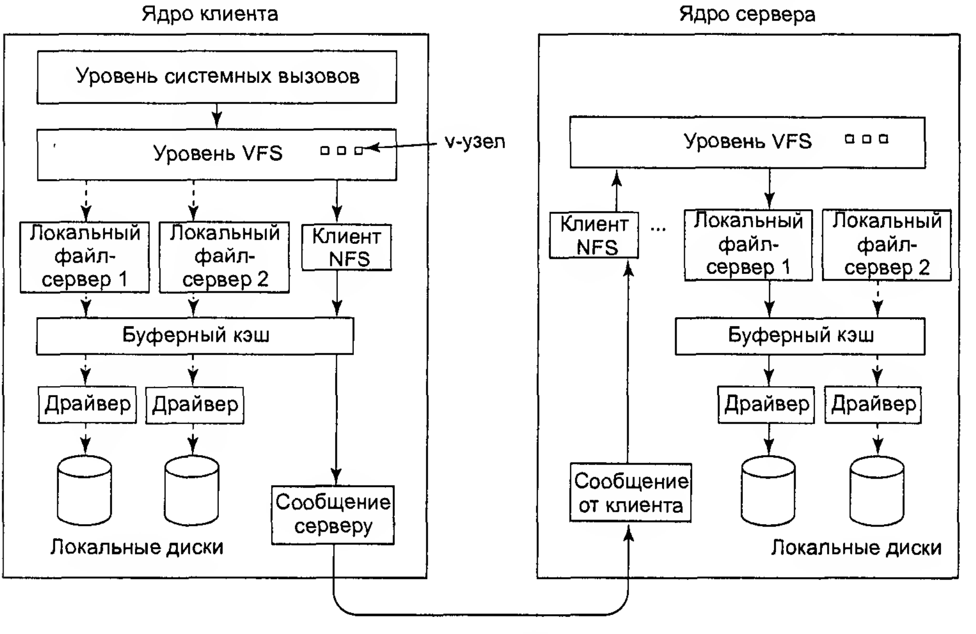


Рис. 10.22. Структура уровней файловой системы NFS

Задача уровня VFS заключается в управлении таблицей, содержащей по одной записи для каждого открытого файла, аналогичной таблице i-узлов для открытых файлов в системе UNIX. В обычной системе UNIX i-узел однозначно указывается

парой (устройство, номер i-узла). Вместо этого уровень VFS содержит для каждого открытого файла записи, называемые **v-узлами** (virtual i-node — виртуальный i-узел). V-узлы используются, чтобы отличать локальные файлы от удаленных. Для удаленных файлов предоставляется информация, достаточная для доступа к ним. Для локальных файлов записываются сведения о файловой системе и i-узле, так как современные системы UNIX могут поддерживать несколько файловых систем (например, V7, Berkeley FFS, ext2fs, /proc, FAT и т. д.). Хотя уровень VFS был создан для поддержки файловой системы NFS, сегодня он поддерживается большинством современных систем UNIX как составная часть операционной системы, даже если NFS не используется.

Чтобы понять, как используются v-узлы, рассмотрим выполнение последовательности системных вызовов mount, open и read. Чтобы смонтировать файловую систему, системный администратор (или сценарий */etc/rc*) вызывает программу *mount*, указывая ей удаленный каталог, локальный каталог, в котором следует смонтировать удаленный каталог, и прочую информацию. Программа *mount* анализирует имя удаленного каталога и обнаруживает имя сервера NFS, на котором располагается удаленный каталог. Затем она соединяется с этой машиной, запрашивая у нее дескриптор удаленного каталога. Если этот каталог существует и его удаленное монтирование разрешено, сервер возвращает его дескриптор. Наконец, программа *mount* обращается к системному вызову mount, передавая ядру полученный от сервера дескриптор каталога.

Затем ядро формирует для удаленного каталога v-узел и просит программу клиента NFS на рис. 10.22 создать в своих внутренних таблицах **r-узел** (удаленный i-узел) для хранения дескриптора файла. V-узел указывает на r-узел. Каждый v-узел на уровне VFS будет в конечном итоге содержать либо указатель на r-узел в программе клиента NFS, либо указатель на i-узел в одной из локальных файловых систем (показанный на рис. 10.22 в виде штриховой линии). По содержимому v-узла можно понять, является ли файл или каталог локальным или удаленным. Если он локальный, то может быть найдена соответствующая файловая система и i-узел. Если файл удаленный, может быть найден удаленный хост и дескриптор файла.

Когда на клиенте открывается удаленный файл, при анализе пути файла ядро обнаруживает каталог, в котором смонтирована удаленная файловая система. Оно видит, что этот каталог удаленный, а в v-узле каталога находит указатель на r-узел. Затем она просит программу клиента NFS открыть файл. Программа клиента NFS просматривает оставшуюся часть пути на удаленном сервере, ассоциированном с монтированным каталогом, и получает обратно дескриптор файла для него. Он создает в своих таблицах r-узел для удаленного файла и докладывает об этом уровню VFS, который помещает в свои таблицы v-узел для файла, указывающий на r-узел. Мы видим, что и в этом случае у каждого открытого файла или каталога есть v-узел, указывающий на r-узел или i-узел.

Вызывающему процессу выдается дескриптор удаленного файла. Этот дескриптор файла отображается на v-узел при помощи таблиц уровня VFS. Обратите внимание, что на сервере не создается никаких записей в таблицах. Хотя сервер готов предоставить дескрипторы файлов по запросу, он не следит за состоянием дескрипторов файлов. Когда дескриптор файла присылается серверу для доступа к файлу,

сервер проверяет дескриптор и использует его, если дескриптор действителен. При проверке может проверяться ключ аутентификации, содержащийся в заголовках вызова удаленной процедуры RPC.

Когда дескриптор файла используется в последующем системном вызове, например read, уровень VFS находит соответствующий v-узел и по нему определяет, является ли он локальным или удаленным, а также какой i-узел или r-узел его описывает. Затем он посылает серверу сообщение, содержащее дескриптор, смещение в файле (хранящееся на стороне клиента, а не сервера) и количество байтов. Для повышения эффективности обмен информацией между клиентом и сервером выполняется большими порциями, как правило, по 8192 байт, даже если запрашивается меньшее количество байтов.

Когда сообщение с запросом прибывает на сервер, оно передается там уровню VFS, который определяет файловую систему, содержащую файл. Затем уровень VFS обращается к этой файловой системе, чтобы прочитать и вернуть байты. Эти данные передаются клиенту. После того как уровень VFS клиента получает 8-килобайтную порцию данных, которую запрашивал, он автоматически посылает запрос на следующую порцию, чтобы она была под рукой, когда понадобится. Такая функция, называемая **опережающим чтением**, позволяет значительно увеличить производительность.

При записи в удаленный файл проходится аналогичный путь от клиента к серверу. Данные также передаются 8-килобайтными порциями. Если системному вызову write подается менее 8 Кбайт данных, данные просто накапливаются локально. Только когда порция в 8 Кбайт готова, она посылается серверу. Если файл закрывается, то весь остаток немедленно посылается серверу.

Кроме того, для увеличения производительности применяется кэширование, как в обычной системе UNIX. Серверы кэшируют данные, чтобы снизить количество обращений к дискам, но это происходит незаметно для клиентов. Клиенты управляют двумя кэшами, одним для атрибутов файлов (i-узлов) и одним для данных. Когда требуется либо i-узел, либо блок файла, проверяется, нельзя ли получить эту информацию из кэша. Если да, то обращения к сети можно избежать.

Хотя кэширование на стороне клиента во много раз повышает производительность, оно также приводит к появлению новых непростых проблем. Предположим, что два клиента сохранили в своих кэшах один и тот же блок файла и что один из них его модифицировал. Когда другой клиент считывает этот блок, он получает из кэша старое значение блока.

Учитывая серьезность данной проблемы, реализация NFS пытается смягчить ее остроту несколькими способами. Во-первых, с каждым блоком кэша ассоциирован таймер. Когда время истекает, запись считается недействительной. Как правило, для блоков с данными таймер устанавливается на 3 с, а для блоков каталога — 30 с. Таким образом, риск несколько снижается. Кроме того, при каждом открытии кэшированного файла серверу посылается сообщение, чтобы определить, когда в последний раз был модифицирован этот файл. Если последнее изменение произошло после того, как была сохранена в кэше локальная копия файла, эта копия из кэша удаляется, а с сервера получается новая копия. Наконец, каждые 30 с истекает время таймера, и все «грязные» (модифицированные) блоки кэша посыла-

ются на сервер. Хотя такая схема и далека от совершенства, подобные «заплатки» позволяют пользоваться этой системой в большинстве практических случаев.

Безопасность в UNIX

Несмотря на свое название, операционная система UNIX была с самого начала многопользовательской системой. Это значит, что безопасность и контроль над информацией были встроены в систему с момента ее создания. В следующих разделах мы рассмотрим некоторые аспекты безопасности в операционной системе UNIX.

Основные понятия

Сообщество пользователей операционной системы UNIX состоит из некоторого количества зарегистрированных пользователей, у каждого из которых есть свой уникальный **UID** (User ID — идентификатор пользователя). UID представляет собой целое число в пределах от 0 до 65 535. Идентификатором владельца помечаются файлы, процессы и другие ресурсы. По умолчанию владельцем файла является пользователь, создавший этот файл, хотя владельца можно сменить.

Пользователи могут организовываться в группы, которые также нумеруются 16-разрядными целыми числами, называемыми **GID** (Group ID — идентификатор группы). Назначение пользователя к группе выполняется вручную системным администратором и заключается в создании нескольких записей в системной базе данных, в которой содержится информация о том, какой пользователь к какой группе принадлежит. Вначале пользователь мог принадлежать только к одной группе, но теперь в некоторых версиях системы UNIX пользователь может одновременно принадлежать к нескольким группам. Чтобы не усложнять вопрос, мы более не станем обсуждать эту возможность.

Основной механизм безопасности в операционной системе UNIX прост. Каждый процесс несет на себе UID и GID своего владельца. Когда создается файл, он получает UID и GID создающего его процесса. Файл также получает набор разрешений доступа, определяемых создающим процессом. Эти разрешения определяют доступ к этому файлу для владельца файла, для других членов группы владельца файла и для всех прочих пользователей. Для каждой из этих трех категорий определяется три вида доступа: чтение, запись и исполнение файла, что обозначается соответственно буквами *r*, *w* и *x* (*read*, *write*, *execute*). Возможность исполнять файл, конечно, имеет смысл только в том случае, если этот файл является исполняемой двоичной[1] программой. Попытка запустить файл, у которого есть разрешение на исполнение, но который не является исполняемым (то есть не начинается с соответствующего заголовка), закончится ошибкой. Поскольку существует три категории пользователей и три вида доступа для каждой категории, все режимы доступа к файлу можно закодировать 9 битами. Некоторые примеры этих 9-разрядных чисел и их значения показаны в табл. 10.14.

[1] Для возможности запуска сценария оболочки, представляющего собой, по сути, не двоичный, а текстовый файл, у этого файла также нужно установить командой *chmod* биты *r* и *x*. — *Примеч. перев.*

Таблица 10.14. Некоторые примеры режимов защиты файлов

Двоичное	Символьное	Рвзрешенный доступ
111000000	rwx------	Владелец может читать, писать и исполнять
111111000	rwxrwx---	Владелец и группа могут читать, писать и исполнять
110100000	rw-r-----	Владелец может читать и писать, группа может читать
110100100	rw-r--r--	Владелец может читать и писать, все остальные могут читать
111101101	rwxr-xr-x	Владелец имеет все права, все остальные могут читать и исполнять
000000000	---------	Ни у кого нет доступа
000000111	------rwx	Только у посторонних есть доступ (странно, но законно)

Первые два примера в таблице понятны. В них к файлу предоставляется полный доступ для владельца файла и для его группы соответственно. В третьем примере группе владельца разрешается читать файл, но не разрешается его изменять, а всем посторонним запрещается всякий доступ. Вариант из четвертого примера часто применяется в тех случаях, когда владелец файла желает сделать файл с данными публичным. Пятый пример показывает режим защиты файла, представляющего собой опубликованную программу. В шестом примере доступ запрещен всем. Такой режим иногда используется для файлов-пустышек, применяемых для реализации взаимных исключений, так как любая попытка создания такого файла приведет к ошибке, если такой файл уже существует. Если несколько программ одновременно попытаются создать такой файл в качестве блокировки, только первой из них это удастся. Режим, показанный в последнем примере, довольно странный, так как он предоставляет всем посторонним пользователям больше привилегий, чем владельцу файла. Тем не менее такой режим допустим. К счастью, у владельца файла всегда есть способ изменить в дальнейшем режим доступа к файлу, даже если ему будет запрещен всякий доступ к самому файлу.

Пользователь, UID которого равен 0, является особым пользователем и называется **суперпользователем** (superuser или **root**). Суперпользователь может читать и писать все файлы в системе, независимо от того, кто ими владеет и как они защищены. Процессы с UID = 0 также обладают возможностью обращаться к небольшой группе системных вызовов, доступ к которым запрещен для обычных пользователей. Как правило, пароль суперпользователя известен только системному администратору, хотя многие студенты младших курсов смотрят на поиск дыр в системе безопасности, которые должны позволить им регистрироваться в системе в качестве суперпользователя, как на увлекательный спорт. Руководство компьютерных центров обычно недовольно такого рода активностью.

Каталоги представляют собой файлы и обладают теми же самыми режимами защиты, что и обычные файлы. Отличие состоит в том, что бит *x* интерпретируется в отношении каталогов как разрешение не исполнения, а поиска в каталоге. Таким образом, каталог с режимом *rwxr-xr-x* позволяет своему владельцу читать, изменять каталог, а также искать в нем файлы, а всем остальным пользователям разрешает только читать каталоги и искать файлы в них, но не создавать в них новые файлы или удалять файлы из этого каталога.

У специальных файлов, соответствующих устройствам ввода-вывода, есть те же самые биты защиты. Благодаря этому может использоваться тот же самый

механизм для ограничения доступа к устройствам ввода-вывода. Например, владельцем специального файла принтера */dev/lp* может быть суперпользователь (root) или специальный пользователь, демон принтера. При этом режим доступа к файлу может быть установлен равным *rw-------*, чтобы все остальные пользователи не могли напрямую обращаться к принтеру. В противном случае при одновременной печати на принтере нескольких процессов получится полный хаос.

Конечно, тот факт, что файлом */dev/lp* владеет демон и этот файл имеет режим доступа *rw-------*, означает, что более никто не может выводить данные на принтер. Хотя такой способ и позволяет избежать множества неприятностей, однако время от времени пользователям бывает необходимо напечатать что-нибудь. В действительности существует более общая проблема регулируемого доступа ко всем устройствам ввода-вывода и другим системным ресурсам.

Эта проблема была решена с помощью добавления к перечисленным выше 9 бит нового бита защиты, **бита SETUID**. Когда выполняется программа с установленным битом SETUID, то запускаемому процессу присваивается не UID вызвавшего его пользователя или процесса, а UID владельца файла. Когда процесс пытается открыть файл, то проверяется его **рабочий UID**, а не UID запустившего его пользователя. Таким образом, если программой, обращающейся к принтеру, будет владеть демон с установленным битом SETUID, тогда любой пользователь сможет запустить ее и запущенный процесс будет обладать полномочиями демона (например, правами доступа к */dev/lp*), но только для запуска этой программы (которая может устанавливать задания в очередь на принтер).

В операционной системе UNIX есть множество программ, владельцем которых является системный администратор, но у них установлен бит SETUID. Например, программе *passwd*, позволяющей пользователям менять свои пароли, требуется доступ записи к файлу паролей. Если разрешить изменять этот файл кому угодно, то ничего хорошего из этой затеи не получится. Вместо этого есть программа, владельцем которой выступает root, и у файла этой программы установлен бит SETUID. Хотя у этой программы есть полный доступ к файлу паролей, она изменит только пароль вызывавшего ее пользователя и не затронет остального содержимого файла.

Помимо бита SETUID, есть также еще и бит SETGID, работающий аналогично и временно предоставляющий пользователю рабочий GID программы. Однако на практике этот бит почти не используется.

Системные вызовы безопасности в UNIX

Лишь небольшое число системных вызовов относятся к безопасности. Наиболее важные системные вызовы перечислены в табл. 10.15. Чаще всего используется системный вызов chmod. С его помощью можно изменить режим защиты файла. Например, оператор

```
s = chmod("/usr/ast/newgame", 0755);
```

устанавливает для файла *newgame* режим доступа *rwxr-xr-x*, что позволяет запускать эту программу всем пользователям (обратите внимание, что 0755 представляет собой восьмеричную константу, что удобно в данном случае, так как биты защиты

группируются тройками). Изменять биты защиты могут только владелец файла и суперпользователь.

Таблица 10.15. Некоторые системные вызовы, относящиеся к безопасности

Системный вызов	Описание
s=chmod(path, mode)	Изменить режим защиты файла
s=access(path, mode)	Проверить разрешение доступа к файлу, используя действительные UID и GID
uid=getuid( )	Получить действительный UID
uid=geteuid( )	Получить рабочий UID
gid=getgid( )	Получить действительный GID
gid=getegid( )	Получить рабочий GID
s=chown(path, owner, group)	Изменить владельца и группу
s=setuid(uid)	Установить UID
s=setgid(gid)	Установить GID

Системный вызов access проверяет, будет ли разрешен определенный тип доступа при заданных UID и GID. Этот системный вызов нужен, чтобы избежать появления брешей в системе безопасности. Он используется в программах с установленным битом SETUID, владельцем которых является root. Такие программы могут выполнять любые действия, поэтому им иногда бывает необходимо определить, уполномочен ли вызвавший их пользователь на выполнение определенных действий. Программа не может просто попытаться получить требуемый доступ, так как любой доступ ей будет обязательно предоставлен.

Следующие четыре системных вызова возвращают значения реального и рабочего UID и GID. К последним трем системным вызовам разрешено обращаться только суперпользователю. Они изменяют владельца файла, а также UID и GID процесса.

Реализация безопасности в UNIX

Когда пользователь входит в систему, программа регистрации *login* (которая является SETUID root) запрашивает у пользователя его имя и пароль. Затем она хэширует пароль и ищет его в файле паролей */etc/passwd*, чтобы определить, соответствует ли хэш-код содержащимся в нем значениям (сетевые системы работают несколько по-иному). Хэширование применяется, чтобы избежать хранения пароля в незашифрованном виде где-либо в системе. Если пароль введен верно, программа регистрации считывает из файла */etc/passwd* имя программы оболочки, которую любит пользователь. Ей может быть программа *sh*, но это также может быть и другая оболочка, например *csh* или *ksh*. Затем программа регистрации использует системные вызовы setuid и setgid, чтобы установить для себя UID и GID (как мы помним, она была запущена как SETUID root). После этого программа регистрации открывает клавиатуру для стандартного ввода (файл с дескриптором 0) и экран для стандартного вывода (файл с дескриптором 1), а также экран для вывода стандартного потока сообщений об ошибках (файл с дескриптором 2).

Наконец, она выполняет оболочку, которую указал пользователь, таким образом, завершая свою работу.

С этого момента начинает работу оболочка с установленными UID и GID, а также стандартными потоками ввода, вывода и ошибок, настроенными на устройства ввода-вывода по умолчанию. Все процессы, которые она запускает при помощи системного вызова fork (то есть команды, вводимые пользователем с клавиатуры), автоматически наследуют UID и GID оболочки, поэтому у них будет верное значение владельца и группы. Все файлы, создаваемые этими процессами, также будут иметь эти значения.

Когда любой процесс пытается открыть файл, система сначала проверяет биты защиты в i-узле файла для заданных значений рабочих UID и GID, чтобы определить, разрешен ли доступ для данного процесса. Если доступ разрешен, файл открывается и процессу возвращается дескриптор файла. В противном случае файл не открывается, а процессу возвращается значение –1. При последующих обращениях к системным вызовам read и write проверка не выполняется. В результате, если режим защиты файла изменяется уже после того, как файл открыт, новый режим не повлияет на процессы, которые уже успели открыть этот файл.

В операционной системе Linux защита файлов и ресурсов осуществляется так же, как и в UNIX. В системе Linux реализованы все функции защиты системы UNIX, и нет почти ничего такого в системе Linux в области защиты, чего нет в операционной системе UNIX.

Резюме

Операционная система UNIX начала свое существование как система разделения времени для мини-компьютеров, но теперь она используется на различных машинах от ноутбуков до суперкомпьютеров. В операционной системе UNIX есть три интерфейса: оболочка, библиотека С и сами системные вызовы. Оболочка позволяет пользователям вводить команды и исполнять их. Это могут быть простые команды, конвейеры или более сложные структуры. Ввод и вывод могут перенаправляться. В библиотеке С содержатся системные вызовы, а также множество расширенных вызовов, например *printf* для записи в файлы форматированного вывода. Фактический интерфейс системных вызовов состоит всего из приблизительно 100 вызовов, каждый из которых выполняет только необходимые функции и ничего более.

К ключевым понятиям операционной системы UNIX относятся процесс, модель памяти, ввод-вывод и файловая система. Процессы могут создавать дочерние процессы, в результате чего формируются деревья процессов. Для управления процессами в UNIX используются две ключевые структуры данных: таблица процессов и структура пользователя. Таблица процессов постоянно находится в памяти, а структура пользователя может выгружаться на диск. При создании процесса дублируется запись в таблице процессов, а также образ памяти процесса. Для планирования применяется алгоритм, основанный на приоритетах, отдающий предпочтение интерактивным процессам.

Модель памяти состоит из трех сегментов для каждого процесса: для текста (исполняемого кода), данных и стека. Изначально для управления памятью использовался свопинг, но в большинстве современных версий системы UNIX для этого применяется страничная подкачка. Состояние каждой страницы отслеживается в карте памяти, а страничный демон поддерживает достаточное количество свободных страниц при помощи алгоритма часов.

Доступ к устройствам ввода-вывода осуществляется при помощи специальных файлов, у каждого из которых есть старший номер устройства и младший номер устройства. Для снижения числа обращений к диску в блочных устройствах ввода-вывода применяется буферный кэш. Для управления кэшем используется алгоритм LRU (Least-Recently-Used — с наиболее давним использованием). Символьный ввод-вывод может осуществляться в обработанном и необработанном режимах. Для дополнительных возможностей символьного ввода-вывода применяются дисциплины линии связи или потоки.

Файловая система в UNIX — иерархическая, с файлами и каталогами. Все диски монтируются в единое дерево каталогов, начинающееся в одном корне. Отдельные файлы могут быть связаны с любым каталогом дерева. Чтобы пользоваться файлом, его нужно сначала открыть. При этом процессу, открывающему файл, возвращается дескриптор файла, который затем используется при чтении этого файла и записи в файл. Внутри файловая система использует три основные таблицы: таблицу дескрипторов файлов, таблицу дескрипторов открытых файлов и таблицу i-узлов. Из этих таблиц таблица i-узлов является наиболее важной. В ней содержится информация, необходимая для управления файлом и позволяющая найти его блоки.

Защита файлов основывается на регулировании доступа для чтения, записи и исполнения, предоставляемого владельцу файла, членам его группы и всем остальным пользователям. Для каталогов бит исполнения интерпретируется как разрешение поиска в каталоге.

Вопросы

1. Когда ядро перехватывает системный вызов, как оно определяет, который системный вызов ему предстоит выполнить?
2. Каталог содержит следующие файлы:

aardvark	feret	Koala	porpoise	unicorn
bonefish	grunion	Llama	quacker	vicuna
capybara	hyena	Marmot	rabbit	weasel
dingo	ibex	Nuthatch	seahorse	Yak
emu	jellyfish	Ostrich	tuna	Zebu

   Какие файлы будут перечислены командой

```
ls [abc]*e*?
```

3. Что делает следующий конвейер оболочки UNIX?

```
grep nd xyz | wc -l
```

4. Напишите конвейер UNIX, печатающий восьмую строку файла *z* в стандартный вывод.
5. Зачем в операционной системе UNIX проводится различие между стандартным выводом и стандартным потоком ошибок, если по умолчанию обоим соответствует терминал?
6. Пользователь вводит с терминала следующие команды:

```
a | b | c&
d | e | f&
```

   Сколько новых процессов будет работать, после того как оболочка обработает эти команды?
7. Когда оболочка UNIX запускает новый процесс, она помещает копии своих переменных окружения, например *HOME*, в стек процесса, чтобы процесс мог определить свой рабочий каталог. Если этот процесс в дальнейшем создаст дочерний процесс, получит ли созданный дочерний процесс эти переменные автоматически?
8. Сколько примерно понадобится времени, чтобы создать дочерний процесс при следующих условиях: размер текста = 100 Кбайт, размер данных = 20 Кбайт, размер стека = 10 Кбайт, размер таблицы процессов = 1 Кбайт, структуры пользователя = 5 Кбайт. Обработка эмулированного прерывания ядром занимает 1 мс, а компьютер может копировать 32-разрядное слово каждые 50 нс. Текстовые сегменты используются совместно.
9. По мере того как мегабайтные программы становились все более распространенными, время, затрачиваемое на обработку системного вызова fork, росло пропорционально росту размеров программ. Что еще хуже, почти все это время просто терялось понапрасну, так как большинство программ сразу после системного вызова fork выполняли системный вызов exec. Чтобы повысить производительность, университет в Беркли разработал новый системный вызов vfork, в котором, вместо того чтобы создавать для дочернего процесса отдельное адресное пространство, адресное пространство используется дочерним процессом совместно с родительским процессом. Опишите ситуацию, в которой неверно работающий дочерний процесс может выполнить действия, делающие семантику системного вызова vfork принципиально отличной от семантики системного вызова fork.
10. Если значение параметра *CPU_usage* (использование центрального процессора) процесса UNIX равно 20, сколько интервалов времени ΔT потребуется, чтобы значение этого параметра уменьшилось до 0, если процесс не получает от планировщика управления? Используйте простой алгоритм затухания, приведенный в тексте.
11. Имеет ли смысл забирать у процесса память, когда процесс переходит в состояние зомби? Почему да или почему нет?
12. С какой аппаратной концепцией тесно связано понятие сигналов? Приведите два примера использования сигналов.

13. Как вы думаете, почему разработчики операционной системы UNIX запретили процессу посылать сигналы процессам, не входящим в его группу процессов?
14. Как правило, системный вызов реализуется при помощи команды эмулированного прерывания. Может ли для этого на компьютере Pentium использоваться обычный процедурный вызов? Если да, то при каких условиях и как? Если нет, то почему?
15. Какие процессы, как правило, обладают более высоким приоритетом, демоны или интерактивные процессы?
16. При создании нового процесса ему должен быть присвоен уникальный номер PID. Достаточно ли для этого хранить в ядре счетчик, увеличивающийся на единицу при создании каждого нового процесса, и использовать этот счетчик как новый PID? Аргументируйте свой ответ.
17. В таблице процессов для каждого процесса хранится PID родительского процесса. Зачем?
18. Какая комбинация битов sharing_flags, используемых командой Linux `clone`, соответствует стандартному системному вызову UNIX `fork`? Созданию потока в UNIX?
19. Планировщик в системе Linux вычисляет «добродетельность» процессов реального времени, добавляя к приоритету число 1000. Можно ли выбрать другую константу так, чтобы алгоритм продолжал выбирать те же процессы?
20. При загрузке операционной системы UNIX (и большинства других операционных систем) начальный загрузчик, хранящийся в 0-м секторе диска, загружает программу загрузки, которая, в свою очередь, загружает операционную систему. Зачем требуется этот лишний промежуточный этап? Ведь было бы проще, если начальный загрузчик, хранящийся в 0-м секторе диска, загружал операционную систему напрямую.
21. Предположим, что текстовый редактор состоит из 100 Кбайт программного кода, 30 Кбайт инициализированных данных и 50 Кбайт BSS. Начальный размер стека составляет 10 Кбайт. Одновременно запускаются три копии этого редактора. Сколько потребуется физической памяти (а) если используется общий текстовый сегмент и (б) если используется общий текстовый сегмент?
22. В 4BSD каждая запись карты памяти содержит индекс для следующей записи в списке свободных страниц, используемый, когда текущее значение находится в списке свободных страниц. Размер поля 16 бит. Размер страниц 1 Кбайт. Влияют ли эти размеры на общий объем памяти, поддерживаемый BSD? Аргументируйте свой ответ.
23. В BSD сегменты данных и стека подкачиваются постранично и выгружаются во временные копии, хранящиеся на специальном диске или дисковом разделе подкачки, но для подкачки текстового сегмента используется сам исполняемый файл. Почему?

24. Опишите способ использования системного вызова mmap и сигналов для создания механизма межпроцессного взаимодействия.
25. Файл отображается на память с помощью системного вызова mmap следующим образом:

```
mmap(65536. 32768. READ. FLAGS. fd. 0)
```

Размер страниц 8 Кбайт. Какой байт файла будет считан при обращении к адресу памяти 72 000?
26. После выполнения системного вызова из предыдущей задачи процесс обращается к системному вызову

```
munmap(65536. 8192)
```

Будет ли он выполнен успешно? Если да, то какие байты файла останутся отображенными на память?
27. Может ли страничное прерывание привести к завершению работы процесса, вызывавшего это прерывание? Если да, приведите пример. Если нет, то почему нет?
28. Возможно ли, чтобы при использовании «приятельской» системы управления памятью два соседних свободных блока одинакового размера сосуществовали и не были объединены в один блок? Если да, приведите пример, как это может произойти. Если нет, докажите, что это невозможно.
29. В тексте утверждалось, что производительность страничной подкачки выше при выгрузке на отдельный раздел диска, а не в файл. Почему это так?
30. Приведите два примера преимущества относительных путей перед абсолютными.
31. Несколько процессов обращается к вызовам блокировки. Скажите, что произойдет при каждом обращении. Если процесс не может получить блокировку, то он блокируется сам.
    а) процесс *A* хочет получить блокировку без монополизации байтов с 0 по 10;
    б) процесс *B* хочет получить блокировку с монополизацией байтов с 20 по 30;
    в) процесс *C* хочет получить блокировку без монополизации байтов с 8 по 40;
    г) процесс *A* хочет получить блокировку без монополизации байтов с 25 по 35;
    д) процесс *B* хочет получить блокировку с монополизацией байта 8.
32. Рассмотрим блокированный файл на рис. 10.16, *в*. Предположим, что процесс пытается получить блокировку байтов 10 и 11 и блокируется. Затем, прежде чем процесс *C* отпустит блокировку, еще один процесс пытается получить блокировку байтов 10 и 11 и также блокируется. Какую проблему добавляет к семантике эта ситуация? Предложите и обоснуйте два решения.
33. Предположим, что происходит обращение к системному вызову lseek с отрицательным значением смещения в файле. Дайте два возможных варианта обработки данной ситуации.
34. Какой доступ получают к файлу его владелец, группа владельца и все остальные пользователи, если режим защиты файла равен 755 (восьмеричное)?

35. У некоторых накопителей на магнитной ленте есть нумерованные блоки и возможность перезаписывать определенные блоки, не затрагивая соседние с ним блоки. Может ли подобное устройство содержать монтированную файловую систему UNIX?
36. На рис. 10.14 после создания связи у Фреда и Лизы есть доступ к файлу *x* в своих каталогах. Является ли доступ к этому файлу абсолютно симметричным, то есть обладают ли оба пользователя одинаковыми правами по отношению к этому файлу?
37. Как было показано, абсолютные пути файлов отсчитываются от корневого каталога, а относительные — от рабочего каталога. Предложите эффективный способ реализации обоих способов поиска файлов.
38. Когда открывается файл */usr/ast/work/f*, требуется несколько обращений к диску, чтобы прочитать i-узел и блоки каталога. Сосчитайте количество необходимых дисковых обращений при условии, что i-узел и корневой каталог постоянно находятся в памяти, а размер всех каталогов один блок.
39. I-узел в системе UNIX содержит 10 дисковых адресов для блоков данных, а также адреса однократного, двукратного и трехкратного косвенных блоков. Чему равен максимальный размер файла в операционной системе UNIX, если каждый из косвенных блоков может содержать 256 дисковых адресов, а размер дискового блока равен 1 Кбайт?
40. Когда при открытии файла с диска считывается i-узел, он помещается в таблицу i-узлов, хранящуюся в памяти. В этой таблице есть поля, отсутствующие на диске. Один из них — это счетчик, отслеживающий количество обращений к i-узлу. Зачем нужно это поле?
41. Почему для управления буферным кэшем применяется алгоритм LRU, в то время как он редко используется для слежения за страницами в системе виртуальной памяти?
42. В операционной системе UNIX есть системный вызов sync, выгружающий модифицированные блоки буферного кэша на диск. При загрузке системы запускается программа *update*. Каждые 30 с она обращается к системному вызову sync, после чего отправляется спать еще на 30 с. Зачем нужна такая программа?
43. Когда операционная система перезагружается после сбоя, как правило, выполняется программа восстановления. Предположим, эта программа обнаруживает, что значение счетчика связей в i-узле равно 2, тогда как только одна каталоговая запись ссылается на данный i-узел. Может ли программа восстановления исправить такую ошибку, и если да, то как?
44. Попытайтесь угадать, который системный вызов UNIX выполняется быстрее всего.
45. Возможно ли удалить связь файла, для которого связь никогда не создавалась? Что произойдет?
46. Основываясь на информации, предоставленной в данной главе, определите, какой максимальный объем данных пользователя можно разместить на

дискете емкостью 1,44 Мбайт, если использовать файловую систему Linux ext2? Предположим, что размер блоков диска равен 1 Кбайт.

47. Учитывая все неприятности, которые могут причинить студенты, если они получат права доступа суперпользователя, можете ли вы сказать, зачем вообще существует понятие суперпользователя?

48. Профессор пользуется общими файлами вместе со своими студентами, помещая их в каталог, к которому предоставлен публичный доступ. Этот каталог расположен в системе UNIX компьютера факультета кибернетики. Однажды профессор спохватывается, что разрешил доступ записи к одному из файлов для всех пользователей. Он изменяет разрешения доступа и убеждается, что файл соответствует оригиналу. На следующий день профессор обнаруживает, что файл был изменен. Как это могло произойти и как это можно было предотвратить?

49. Напишите минимальную оболочку, способную выполнять простые команды. Она также должна быть способна запускать эти команды в фоновом режиме.

50. С помощью ассемблера и системных вызовов BIOS напишите программу, загружающуюся с гибкого диска на компьютере с процессором Pentium. Эта программа должна использовать вызовы BIOS для чтения ввода с клавиатуры и вывода эха вводимых символов на экран, просто чтобы продемонстрировать, что она работает.

51. Напишите программу для неинтеллектуального терминала, позволяющую соединить две рабочие станции, управляемые операционными системами UNIX или Linux, через последовательные порты. Используйте системные вызовы стандарта POSIX для настройки портов.

Глава 11
Рассмотрение конкретных случаев: Windows 2000

Windows 2000 — это современная операционная система, работающая на настольных персональных компьютерах старших моделей и серверах. Данная глава посвящается различным аспектам этой системы. Мы начнем ее изучение с истории, затем перейдем к архитектуре системы. После этого мы рассмотрим процессы, управление памятью, ввод-вывод, файловую систему и, наконец, систему безопасности. Мы не станем рассматривать сетевые аспекты, так как одна эта тема легко может растянуться на целую главу или даже целую книгу.

История Windows 2000

Операционные системы корпорации Microsoft для настольных и переносных компьютеров можно разделить на три семейства: MS-DOS, Consumer Windows (Windows 95/98/Me) и Windows NT. Ниже мы кратко опишем каждое из этих семейств.

MS-DOS

В 1981 году корпорация IBM, в то время крупнейшая и мощнейшая компьютерная компания мира, создала персональный компьютер IBM PC, основанный на процессоре Intel 8088. Персональный компьютер был оснащен 16-разрядной однопользовательской операционной системой реального режима с командной строкой, названной MS-DOS 1.0. Эта операционная система поставлялась крохотной начинающей фирмой Microsoft, большей частью известной в те годы как создатель интерпретатора BASIC для систем на базе процессоров Intel 8080 и Zilog Z80. Операционная система состояла из резидентной программы размером в 8 Кбайт, довольно близко копирующей CP/M, примитивную операционную систему для 8-разрядных процессоров Intel 8080 и Zilog Z80. Два года спустя была выпущена более мощная операционная система MS-DOS 2.0, состоящая из 24 Кбайт резидентного кода. Она содержала программу обработки командной строки (оболочку) с большим количеством функций, позаимствованных у операционной системы UNIX.

Когда фирма Intel выпустила 286-й процессор, корпорация IBM создала на его основе новый компьютер, PC/AT, выпущенный в 1986 году. Буквы AT означали

Advanced Technology (передовая технология), так как процессор Intel 286 работал на впечатляющей тогда частоте в 8 МГц и мог адресоваться (правда, с большим трудом) к 16 Мбайт памяти. На практике у большинства систем было максимум 1 Мбайт или 2 Мбайт, поскольку память в те времена стоила очень дорого. Персональный компьютер IBM PC/AT поставлялся вместе с операционной системой MS-DOS 3.0 фирмы Microsoft, занимавшей в памяти 36 Кбайт. С годами в операционной системе MS-DOS появилось много новых функций, но она по-прежнему оставалась операционной системой, ориентированной на командную строку.

Windows 95/98/Me

Вдохновленная пользовательским интерфейсом компьютера Apple Lisa, предшественника Apple Macintosh, корпорация Microsoft решила добавить к операционной системе MS-DOS графический интерфейс пользователя (оболочку), которую она назвала **Windows**. Операционная система Windows 1.0, выпущенная в 1985 году, была чем-то вроде суррогата. Версия Windows 2.0, разработанная для PC/AT и выпущенная в 1987 году, была не намного лучше. Наконец, операционная система Windows 3.0 для персонального компьютера с центральным процессором Intel 386 (выпущенная в 1990 году), и особенно последовавшие за ней версии 3.1 и 3.11 добились большого коммерческого успеха. Ни одна из этих ранних версий Windows не являлась настоящей операционной системой, скорее графическим интерфейсом пользователя поверх MS-DOS, которая продолжала управлять машиной и файловой системой. Все программы работали в одном и том же адресном пространстве, и ошибка в одной из них могла повесить всю систему.

Выход в августе 1995 года Windows 95 до сих пор не привел к полному вытеснению системы MS-DOS, хотя почти все функции MS-DOS были перенесены в Windows. Как Windows 95, так и новая версия MS-DOS 7.0 содержали большинство особенностей монолитной операционной системы, включая виртуальную память и управление процессами. Однако операционная система Windows 95 не была полностью 32-разрядной программой. Она содержала большие куски 16-разрядного ассемблерного кода (а также немного 32-разрядного) и продолжала использовать файловую систему MS-DOS, практически со всеми ее ограничениями. Единственное значительное изменение файловой системы заключалось в добавлении длинных имен файлов к именам из 8 + 3 символа, разрешенным в MS-DOS.

Даже в выпуске Windows 98 в июне 1988 года MS-DOS все еще присутствовала (теперь она называлась версией 7.1) и состояла из 16-разрядного кода. Хотя теперь еще больше функций было переведено из MS-DOS-части системы в часть Windows, а поддержка больших дисковых разделов стала стандартом, по своему строению операционная система Windows 98 не сильно отличалась от Windows 95. Основное отличие заключалось в интерфейсе пользователя, в большей степени интегрировавшем в себе Интернет и рабочий стол пользователя. Именно эта интеграция и привлекла внимание Министерства юстиции США, которое затем выдвинула против корпорации Microsoft иск, обвиняя корпорацию Microsoft в нарушении закона о монополиях. Корпорация Microsoft яростно отрицала свою вину. В апреле 2000 года Федеральный суд США согласился с правительством.

Кроме того, что в ядре операционной системы Windows 98 содержался большой кусок 16-разрядного ассемблерного кода, у этой системы были еще две серьезные проблемы. Во-первых, хотя эта система была многозадачной, само ядро не было реентерабельным. Если процесс был занят управлением какой-либо структурой данных в ядре, а затем его квант времени заканчивался и начинал работу другой процесс, новый процесс мог получить структуру данных в противоречивом состоянии. Чтобы предотвратить возникновение подобной проблемы, большинство процессов, зайдя в ядро, первым делом получали гигантский мьютекс, покрывающий всю систему, прежде чем приступить к каким-либо действиям. Хотя такой подход и устранял потенциальную угрозу противоречивости структур данных, он также уничтожал большую часть преимуществ многозадачности, так как процессам, чтобы войти в ядро, часто приходилось ждать, пока другой процесс ядро покинет.

Во-вторых, у каждого процесса было 4-гигабайтное адресное пространство, в котором первые 2 Гбайт полностью принадлежали процессу. Однако следующий 1 Гбайт совместно использовался (с возможностью записи) всеми процессами системы. Нижний 1 Мбайт также совместно использовался всеми процессами, чтобы все они могли получать доступ к векторам прерывания MS-DOS. Эта возможность вовсю использовалась большинством приложений Windows 98. В результате ошибка в одной программе могла повредить ключевые структуры данных, используемые посторонними процессами, вследствие чего все эти процессы рушились. Что еще хуже, последний 1 Гбайт совместно использовался (с возможностью записи) процессами и ядром и содержал некоторые критические структуры данных. Любая программа, записав поверх этих структур какой-либо мусор (преднамеренно или нет), могла вывести из строя всю систему. Очевидное решение, заключающееся в том, чтобы не помещать структуры данных ядра в пространство пользователя, было неприменимо, так как старые программы, написанные для MS-DOS, не смогли бы тогда работать в Windows 98.

В 2000 году корпорация Microsoft выпустила слегка измененную версию системы Windows 98, названную **Windows Me** (Windows Millennium Edition — Windows, выпуск тысячелетия). Хотя в данной версии были исправлены некоторые ошибки, а также добавлены новые функции, под внешней оболочкой скрывалась все та же Windows 98. Новые функции включали в себя улучшенные возможности организации и совместного использования изображений, музыки и фильмов, серьезнее поддерживали работу с сетью на дому и многопользовательские игры, а также содержали больше функций, относящихся к Интернету, таких как поддержка мгновенных сообщений и широкополосных соединений (кабельных модемов и ADSL). Одна интересная новая функция состояла в возможности восстановить прежние настройки компьютера после неверной установки каких-либо параметров. Если пользователь перенастраивал систему (например, изменял разрешение экрана с 640×480 на 1024×768), и после этого система переставала работать, теперь он мог вернуться к последней работающей конфигурации.

Windows NT

К концу 80-х корпорация Microsoft осознала, что построение современной 32-разрядной операционной системы поверх 16-разрядной системы MS-DOS представля-

ет собой не лучшее решение. Компания Microsoft наняла Дэвида Катлера, одного из ключевых разработчиков операционной системы VMS, созданной корпорацией DEC, и поручила ему возглавить работу над совершенно новой 32-разрядной операционной системой, совместимой с Windows. Эта новая система, названная позднее **Windows NT** (буквы NT означали New Technology — новая технология), предназначалась для деловых приложений, решающих критически важные, ответственные задачи, а также для домашнего использования. В это время мэйнфреймы все еще правили деловым миром, поэтому предположение, что компании будут использовать персональные компьютеры для чего-либо важного, тогда выглядело довольно утопично. Тем не менее, как показала история, это был правильный выбор. Такие свойства, как безопасность и высокая надежность, отсутствовавшие в версиях Windows, основанных на MS-DOS, были поставлены в данном проекте во главу угла. Опыт работы с VMS, полученный Катлером, отчетливо проявлялся при создании системы, и в строении NT и VMS есть нечто большее, чем просто поверхностное сходство.

Проект оказался успешным, и в 1993 году была выпущена первая версия, названная Windows NT 3.1. Начальный номер версии был выбран так, чтобы он соответствовал номеру версии популярной тогда 16-разрядной Windows 3.1. Корпорация Microsoft ожидала, что операционная система NT быстро вытеснит Windows 3.1, так как по формальным показателям NT значительно превосходила ее.

К большому удивлению разработчиков, почти все пользователи предпочли остаться на уже знакомой им старой 16-разрядной версии, а не переходить на неизвестную 32-разрядную систему, какой бы хорошей она ни была. Для операционной системы NT требовалось значительно больше памяти, чем для Windows 3.1, к тому же для новой системы не было 32-разрядных программ, поэтому зачем нужны были пользователям все эти хлопоты? Поскольку операционная система NT 3.1 потерпела неудачу на рынке, корпорация Microsoft решила выпустить 32-разрядную версию Windows 3.1, а именно Windows 95. Пользователи продолжали упорствовать, не желая переходить на NT, и корпорация Microsoft выпустила Windows 98 и, наконец, Windows Me. О каждой из которых заявлялось, что это самый последний выпуск операционной системы, основанной на MS-DOS.

Несмотря на тот факт, что почти все покупатели и большинство корпораций проигнорировало операционную систему NT 3.1 для настольных систем, эта операционная система стала пользоваться некоторым спросом на рынке серверов. В 1994 и 1995 годах было выпущено несколько новых 3.x версий с небольшими изменениями. Эти версии начали также медленно приобретать сторонников среди пользователей настольных машин.

Первое значительное усовершенствование системы NT появилось в 1996 году в виде версии NT 4.0. Эта система обладала мощностью, безопасностью и надежностью современной операционной системы, но она также использовала тот же самый пользовательский интерфейс, что и очень популярная тогда Windows 95. Эта совместимость облегчала пользователям переход с Windows 95 на NT, и многие пользователи так и поступили. Некоторые отличия между Windows 95/98 и Windows NT приведены в табл. 11.1.

Таблица 11.1. Некоторые отличия между Windows 95/98 и Windows NT

Аспект	Windows 95/98	Windows NT
Полностью 32-разрядная система?	Нет	Да
Безопасность?	Нет	Да
Защищенное отображение файлов?	Нет	Да
Приватное адресное пространство для каждой программы MS-DOS?	Нет	Да
Unicode?	Нет	Да
Процессор	Intel 80x86	80x86, Alpha, MIPS, ...
Многопроцессорная поддержка?	Нет	Да
Реентерабельность кода операционной системы?	Нет	Да
Plug and play?	Да	Нет
Управление питанием?	Да	Нет
Файловая система FAT-32?	Да	По желанию
Файловая система NTFS?	Нет	Да
Win32 API?	Да	Да
Поддержка всех старых программ MS-DOS?	Да	Нет
Критические данные ОС, доступные пользователю для записи?	Да	Нет

С самого начала операционная система NT разрабатывалась в расчете на переносимость системы на другие платформы, поэтому она была практически полностью написана на С с очень небольшими включениями на ассемблере для низкоуровневых функций, таких как обработка прерываний. Первый выпуск состоял из 3,1 млн строк на С для операционной системы, библиотек и подсистем окружения (они будут обсуждаться ниже). Когда вышла NT 4.0, программная основа выросла до 16 млн строк кода, все так же большей частью на языке С, хотя для написания пользовательского интерфейса было использовано некоторое количество C++. К этому времени система обладала высокой переносимостью, различные ее версии работали на компьютерах с процессорами Pentium, Alpha, MIPS и Power PC, а также некоторых других. С тех пор некоторые из этих версий перестали поддерживаться. История развития NT приведена в [367]. В этой книге также много рассказывается о ключевых участниках этой истории.

Windows 2000

Следом за NT 4.0 предполагалось выпустить версию NT 5.0. Однако в 1999 году корпорация Microsoft изменила ее название на Windows 2000, в основном из-за попыток найти нейтральное имя, выглядящее логическим продолжением как для пользователей Windows 98, так и для пользователей NT. Таким образом, корпорация Microsoft рассчитывала иметь единую операционную систему, построенную на основе надежной 32-разрядной технологии, но использующую популярный интерфейс пользователя системы Windows 98.

Поскольку в действительности операционная система Windows 2000 представляет собой NT 5.0, она унаследовала множество свойств системы NT 4.0. Она является полностью 32-разрядной (планируется переход на 64-разрядную) мно-

гозадачной системой с индивидуально защищенными процессами. У каждого процесса есть собственное 32-разрядное (будет 64-разрядное) виртуальное адресное пространство. Операционная система работает в режиме ядра, тогда как процессы пользователя работают в пользовательском режиме, что обеспечивает полноценную защиту (в отличие от Windows 98). У процессов может быть один или несколько потоков, видимых для операционной системы и управляемых ею. Она удовлетворяет требованиям безопасности уровня C2 Министерства обороны США для всех файлов, каталогов и процессов, а также других объектов, которые могут использоваться совместно (по крайней мере, если гибкий диск вынут, а сеть отключена). Наконец, она обладает полной поддержкой симметричных многопроцессорных систем с числом процессоров от 2 до 32.

Тот факт, что Windows 2000 в действительности представляет собой NT 5.0, проявляется во многом. Например, системный каталог называется *\winnt*, а двоичный файл операционной системы (в каталоге *\winnt\system32*) называется *ntoskrnl.exe*. Если щелкнуть на этом файле правой кнопкой мыши и посмотреть его свойства, мы увидим, что номер его версии представляет собой *5.xxx.yyy.zzz*, где *5* означает NT 5, *xxx* — номер выпуска, *yyy* — номер сборки (компиляции), а *zzz* — дополнительный номер версии. Кроме того, многие файлы в каталоге *\winnt* и его подкаталогах содержат буквы *nt* в своих именах, как, например, виртуальный эмулятор MS-DOS *ntvdm*.

Операционная система Windows 2000 — это не просто улучшенная версия NT 4.0 с интерфейсом Windows 98. Начнем с того, что она содержит множество других функций, которые ранее были только в Windows 98. К ним относится полная поддержка устройств plug-and-play, шины USB, стандарта IEEE 1394 (FireWire), IrDA (Infrared Data Association — стандарт на инфракрасную передачу данных и вывод на печать, разработанный ассоциацией IrDA), а также, среди прочего, управление питанием. Кроме того, были добавлены несколько новых функций, не присутствовавших ранее в других операционных системах корпорации Microsoft, включая каталоговую службу Active Directory, систему безопасности Kerberos, поддержку смарт-карт, инструменты мониторинга системы, лучшую интеграцию лэптопов и настольных компьютеров, инфраструктуру системного администрирования и рабочие объекты. Другая новая особенность файловой системы NTFS состоит в разновидности связи с копированием при записи, при использовании которой два пользователя могут совместно использовать один связанный файл. Как только один из пользователей начинает запись в этот файл, автоматически создается копия файла.

Еще одно значительное усовершенствование заключается в интернационализации. Операционная система NT 4.0 поставлялась в виде отдельных версий для различных языков, так как текстовые строки были внедрены в программный код. При установке английского программного пакета на голландский компьютер часто части операционной системы переставали использовать голландский язык и переходили на английский, поскольку определенные файлы, содержащие программные и текстовые строки, были перезаписаны. Эта проблема[1] была устранена. Опера-

[1] В действительности проблема несколько серьезнее. В конце концов, иностранный язык можно и выучить. Дело в том, что в операционных системах Windows (как 98, так и NT) многие пакеты поставляются вместе с библиотеками, на тот случай, если они не установлены в системе. Большинство же библиотек хранится в одном общем каталоге. Иногда неверно написанная программа установки пакета заменяет какой-либо DLL-файл более старой версией, после чего перестают работать другие пакеты. — *Примеч. перев.*

ционная система Windows 2000 состоит из единого двоичного кода, работающего во всех странах мира. Для каждой установки системы и даже для каждого пользователя можно выбрать язык, который будет использоваться во время работы системы. Это возможно потому, что все пункты меню, строки диалоговых окон, сообщения об ошибках и другие текстовые строки были удалены из операционной системы и помещены в специальные каталоги, по одному для каждого языка. Как и предыдущие версии операционной системы NT, Windows 2000 использует кодировку Unicode для поддержки языков, не использующих латинский алфавит, например русского, греческого, иврита и японского.

Единственная вещь, которой нет в Windows 2000 — это MS-DOS. Ее просто нет здесь ни в каком виде (как не было в NT). Есть интерфейс командной строки, но это новая 32-разрядная программа, включающая функциональность старой системы MS-DOS, а также некоторые новые функции[1].

Несмотря на многочисленные свойства, способствующие переносимости системы с точки зрения программ, аппаратуры, языков и т. д., в одном отношении операционная система Windows 2000 обладает меньшей переносимостью, чем NT 4.0. Она работает только на двух платформах — Pentium и Intel IA-64[2]. Изначально операционная система NT поддерживала дополнительные платформы, включая PowerPC, MIPS и Alpha, но с годами корпорация Microsoft перестала поддерживать эти процессоры один за другим по коммерческим соображениям.

Как и предыдущие версии NT, в настоящее время Windows 2000 поставляется в виде нескольких уровней продукта: Professional, Server, Advanced server и Datacenter Server. Однако различия между этими версиями незначительны, и во всех версиях используется один и тот же исполняемый двоичный код. При установке системы тип продукта записывается во внутренней базе данных (системном реестре). Во время загрузки операционная система проверяет содержимое реестра, определяя версию программного продукта. Различия между ними показаны в табл. 11.2.

Таблица 11.2. Различные версии Windows 2000

Версия	Максимальный размер ОЗУ, Гбайт	CPU	Максимальное число клиентов	Размер кластера	Оптимизация
Professional	4	2	10	0	Время отклика
Server	4	4	Не ограничено	0	Пропускная способность
Advanced server	8	8	Не ограничено	2	Пропускная способность
Datacenter server	64	32	Не ограничено	4	Пропускная способность

Как видно из таблицы, различия включают максимальный размер поддерживаемой оперативной памяти, максимальное количество центральных процессоров (для многопроцессорной конфигурации) и максимальное число клиентов, кото-

[1] 16-разрядного кода в NT действительно нет, но это не мешает запускать в NT и в Windows 2000 большинство старых 16-разрядных программ, написанных для MS-DOS и Windows 3.1. Для этого в системе содержится специальная система эмуляции 16-разрядной машины. — *Примеч. перев.*

[2] Пока что существует лишь в виде эмулятора. — *Примеч. перев.*

рые могут быть обслужены данной системой в качестве сервера. Размер кластера означает способность операционной системы Windows 2000 представить для окружающего мира две или четыре отдельные машины в виде одного сервера, что часто бывает полезно, например, для web-серверов. Наконец, на Windows 2000 Professional по-другому настраиваются параметры по умолчанию. В этой системе интерактивным процессам предоставляется преимущество перед пакетными заданиями, хотя это можно изменить, если необходимо. Последнее отличие заключается в том, что на серверах, в отличие от Windows 2000 Professional, предоставляется дополнительное программное обеспечение, а с системой Windows 2000 Datacenter server поставляются дополнительные средства управления большими заданиями.

Причина существования нескольких версий является исключительно коммерческой: это позволяет корпорации Microsoft получать с крупных компаний больше денег, чем с индивидуальных клиентов, за практически один и тот же программный продукт. Идея эта не нова и корпорация Microsoft не уникальна в применении такой рыночной тактики. Уже много лет авиалинии берут с деловых пассажиров за полет тем же рейсом значительно большую сумму. Причем не только за бизнес-класс, но также и за возможность покупки билета за день до полета вместо традиционного заказа за месяц.

Формально различием в версиях управляют в нескольких местах программы всего две переменные, считываемые из реестра: *ProductType* и *ProductSuite*. В зависимости от их значений выполняется слегка отличный код. Изменение значений этих переменных рассматривается как нарушение лицензии. Кроме того, система перехватывает любые попытки изменить их и регистрирует эти попытки нестираемым способом, так что впоследствии можно доказать факт нарушения лицензии.

Кроме основной операционной системы, корпорация Microsoft также разработала несколько инструментальных программ для продвинутых пользователей. К этим программам относятся Support Tools, Software Development Kit, Driver Development Kit и Resource Kit. Это большие наборы утилит для отладки и мониторинга системы. Инструментарий поддержки распространяется на компакт-диске Windows 2000 в каталоге *\support\tools*. Файлы не устанавливаются стандартной процедурой, но их можно установить специальной программой *setup.exe*, расположенной в этом же каталоге. SDK и DDK разработчики могут получить в Интернете по адресу *msdn.microsoft.com*. Resource Kit представляет собой розничный продукт корпорации Microsoft. Кроме того, существует множество утилит для слежения за внутренней работой Windows 2000, разработанных другими компаниями. Например, прекрасный набор инструментов можно бесплатно скачать с web-сайта www.sysinternals.com. Некоторые из этих программ даже предоставляют больше информации, чем соответствующие инструменты корпорации Microsoft.

Windows 2000 представляет собой чрезвычайно сложную систему, на сегодняшний день состоящую более чем из 29 млн строк на С. Если распечатать это по 50 строк на странице и сброшюровать по 1000 страниц в книге, то полный код займет 580 томов. Для такого опуса потребуется 23 погонных метра полочного пространства (для версии в мягкой обложке). Если же эти книжные полки расположить в виде книжных шкафов, шириной в 1 м и в 6 полок высотой, то понадобится 4 таких шкафа, чтобы вместить распечатанный исходный текст системы.

Шутки ради в табл. 11.3 приведены данные об объеме исходного текста различных операционных систем. (В первой строке каждой ячейки таблицы содержится версия системы, вторая строка представляет собой количество строк исходного текста, где K = 1000, а M = 1 000 000.) Однако следует очень осторожно сравнивать различные операционные системы в этой таблице, так как то, что составляет операционную систему, отличается от системы к системе. Например, вся оконная система и графический интерфейс пользователя являются частью ядра в Windows, но не входят в ядро ни в одной из версий системы UNIX. В них это просто пользовательский процесс. С учетом X Windows ко всем версиям UNIX добавится 1,5 млн строк кода, а ведь при этом даже не учитывается исходный текст графического интерфейса пользователя (Motif, GNOME и т. д.), который также не является частью операционной системы UNIX. Кроме того, некоторые системы содержат код для различных архитектур (например, пять для 4.4BSD и девять для Linux), при этом каждая архитектура добавляет от 10 000 до 50 000 строк кода. Причина, по которой операционная система Free BSD 1.0 состоит всего лишь из 235 000 строк кода, тогда как 4BSD Lite, от которой она произошла, состояла из 743 000 строк, заключается в том, что из Free BSD была выброшена поддержка всех устаревших архитектур (например, VAX).

Таблица 11.3. Сравнение размеров некоторых операционных систем

Год	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9 K					
1979	V7 21 K					
1980		4.1 38 K				
1982	Sys III 58 K					
1984		4.2 98 K				
1986		4.3 179 K				
1987	SVR3 92 K		1.0 13 K			
1989	SVR4 280 K					
1991				0.01 10 K		
1993		Free 1.0 235 K			5.3 850 K	3.1 6 M
1994		4.4 Lite 743 K		1.0 165 K		3.5 10 M
1996				2.0 470 K		4.0 16 M
1997			2.0 62 K		5.6 1,4 M	
1999				2.2 1 M		
2000		Free 4.0 1,4 M			5.8 2,0 M	2000 29 M

В различных системах также очень сильно разнится количество файловых систем, драйверов устройств и поставляемых библиотек. К тому же система Windows содержит большое количество тестового кода, которого не содержит UNIX, некоторые утилиты и поддержку большого числа других языков, помимо английского. Наконец, эти измерения производились различными людьми, что тоже повлияло на разницу в учете (например, следует ли учитывать сборочные файлы проектов, заголовочные файлы и документацию?). Таким образом, это больше напоминает сравнение яблок не с апельсинами, а с телефонными аппаратами. Однако все учет-

ные данные внутри одного семейства операционных систем поступали из одного источника, поэтому сравнение внутри одного семейства имеет некоторый смысл.

Несмотря на все вышесказанное, два вывода очевидны:

1. Распухание операционных систем, похоже, так же неотвратимо, как смерть и налоги.
2. Система Windows значительно превосходит UNIX в размерах.

Что считать лучшим — большие или компактные системы, — до сих пор остается предметом жарких споров. Аргумент в пользу компактных систем заключается в том, что концепция небольших размеров и принцип «ничего лишнего» приводит к созданию управляемых, надежных систем, понятных для пользователя. В пользу больших систем приводится обычно тот аргумент, что пользователю, мол, требуется много различных функций. В любом случае должно быть ясно, что студенты, планирующие написать с нуля полноценную, современную операционную систему, берут на себя совершенно непосильную задачу.

Хотя Windows 2000 уже сейчас является чемпионом мира в тяжелом весе, если считать чистую массу, эта операционная система все продолжает расти, ошибки устраняются, а новые функции добавляются. Довольно интересен способ, которым корпорация Microsoft управляет разработкой операционной системы. Сотни программистов целый день работают над различными аспектами Windows 2000. Когда кусок программы закончен, программист посылает его по сети команде сборщиков. Каждый день в 18:00 дверь закрывается и система собирается заново (то есть перекомпилируется и компонуется). Каждая собранная таким образом версия системы получает порядковый номер, который можно увидеть в параметрах файла *ntoskrnl.exe* (первый опубликованный выпуск системы Windows 2000 имел номер 2195).

Затем новая операционная система, опять же по сети, распространяется на тысячи машин кампуса корпорации Microsoft в Редмонде, штат Вашингтон, где ее всю ночь подвергают интенсивному тестированию. Ранним утром следующего дня результаты тестирования рассылаются соответствующим группам, чтобы они могли видеть, как работают их новые программы. Затем каждая группа программистов решает, над какой программой они будут работать в этот день. Затем снова наступает 18:00, и весь цикл повторяется.

Программирование в Windows 2000

Теперь настала пора начать изучение технических аспектов операционной системы Windows 2000. Однако, прежде чем приступить к рассмотрению деталей внутренней структуры системы, обсудим программный интерфейс и реестр, небольшую базу данных, хранящуюся в памяти системы.

Программный интерфейс Win32 API

Как и в других операционных системах, в Windows 2000 есть свой набор системных вызовов, которые она может выполнять. Однако корпорация Microsoft никогда не публиковала список системных вызовов Windows, кроме того, она постоянно меня-

ет их от одного выпуска к другому. Вместо этого корпорация Microsoft определила набор функциональных вызовов, называемый **Win32 API** (Win32 Application Programming Interface — интерфейс прикладного программирования). Эти вызовы опубликованы и полностью документированы. Они представляют собой библиотечные процедуры, которые либо обращаются к системным вызовам, чтобы выполнить требуемую работу, либо, в некоторых случаях, выполняют работу прямо в пространстве пользователя. Существующие вызовы Win32 API не изменяются с новыми выпусками системы Windows, хотя часто добавляются новые вызовы Win32 API.

Двоичные программы для процессоров Intel x86, строго придерживающиеся интерфейса Win32 API, будут без каких-либо изменений работать на всех версиях Windows, начиная с Windows 95. Как показано на рис. 11.1, для операционной системы Windows 3.1 требуется дополнительная библиотека, преобразующая подмножество 32-разрядных вызовов Win32 API в 16-разрядные системные вызовы, но для остальных систем никакой адаптации не требуется. Следует отметить, что в операционной системе Windows 2000 к интерфейсу Win32 API добавлено значительное количество новых функций, поэтому в ней есть дополнительные вызовы API, не включенные в старые версии Win32, которые не будут работать на старых версиях Windows.

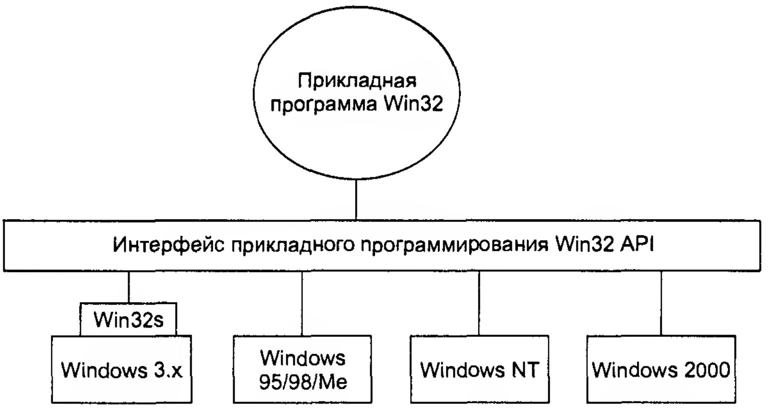


Рис. 11.1. Интерфейс Win32 API позволяет программам работать почти на всех версиях Windows

Философия Win32 API полностью отлична от философии UNIX. В операционной системе UNIX все системные вызовы опубликованы и формируют минимальный интерфейс: удаление даже одного из них приведет к снижению функциональности операционной системы. Философия Win32 заключается в предоставлении всеобъемлющего интерфейса, часто с возможностью выполнить одно и то же тремя или четырьмя способами, включающего множество функций (например, процедур). Эти функции, очевидно, не должны быть (и не являются) системными вызовами, как, например, вызов API для копирования целого файла.

Многие вызовы Win32 API создают объекты ядра того или иного типа, например файлы, процессы, потоки, каналы и т. д. Каждый вызов, создающий объект, возвращает вызывающему процессу результат, называемый дескриптором. Этот

манипулятор может использоваться впоследствии для выполнения операций с объектом. Дескриптор специфичен для процесса, создавшего этот объект. Он не может быть просто передан другому процессу и использован там (так же как дескрипторы файлов в системе UNIX не могут передаваться другим процессам). Однако при определенных обстоятельствах дескриптор может быть дублирован и передан другому процессу защищенным способом, что предоставляет второму процессу контролируемый доступ к объекту, принадлежащему первому процессу. С каждым объектом ассоциирован дескриптор безопасности, подробно описывающий, кто и какие действия может, а какие не может выполнять с данным объектом.

Не все создаваемые системой структуры данных являются объектами, а также не все объекты являются объектами ядра. Только настоящие объекты ядра должны иметь имена, защиту и могут совместно использоваться каким-либо образом. У каждого объекта ядра есть определенный системой тип, четко определенный набор операций, которые могут быть выполнены с объектом, и определенное место хранения в ядре. Хотя пользователи могут выполнять операции с объектом (при помощи вызовов Win32 API), но они не могут напрямую обращаться к объекту.

Сама операционная система может также создавать и использовать объекты, чем она активно занимается. Большинство этих объектов создаются, чтобы позволить одному компоненту системы некоторое время хранить определенную информацию или чтобы передать некоторую структуру данных другому компоненту системы. Например, при загрузке драйвера создается объект, в котором хранятся его свойства и указатели на содержащиеся в нем функции. Затем операционная система обращается к драйверу, используя этот объект.

Иногда говорят, что система Windows 2000 является объектно-ориентированной, так как единственный способ управления объектом заключается в вызове операций, связанных с дескриптором объекта, путем обращения к вызовам Win32 API. С другой стороны, в этой схеме отсутствуют основные свойства объектно-ориентированной системы, такие как наследование и полиморфизм.

Вызовы Win32 API покрывают все мыслимые области, с которыми может работать операционная система, и довольно много областей, с которыми операционная система, по идее, работать не должна. Естественно, этот интерфейс содержит вызовы для создания процессов и потоков и для управления ими. Также существует множество вызовов, относящихся к межпроцессному (в действительности межпоточному) взаимодействию. Это такие вызовы, как создание, уничтожение и использование мьютексов, семафоров, событий и других объектов IPC (InterProcess Communication — межпроцессное взаимодействие).

Хотя большая часть системы управления памятью невидима для программистов (по сути, она представляет собой просто страничную подкачку по требованию), одна важная функция видима, а именно способность процесса отображать файл на свою виртуальную память. Это предоставляет процессу возможность читать и писать части файла, как если бы они представляли собой просто слова в памяти.

Важной частью многих программ является файловый ввод-вывод. С точки зрения Win32, файл представляет собой просто линейную последовательность байтов. Интерфейс Win32 предоставляет более 60 вызовов для создания и уничтожения файлов и каталогов, открытия и закрытия файлов, их чтения и записи, чтения и изменения атрибутов файлов и многого другого.

Другой областью, в которой интерфейс Win32 предоставляет вызовы, является безопасность. У каждого процесса есть идентификатор, сообщающий о процессе, кто он есть. А у каждого процесса может быть список управления доступом, в котором очень подробно сообщается, какие пользователи имеют к нему доступ и какие операции они могут выполнять с этим объектом. Такой подход обеспечивает высокую степень детализации настроек параметров безопасности, в которых можно разрешить или запретить определенный тип доступа к каждому объекту для индивидуальных пользователей или групп.

Процессы, потоки, синхронизация, управление памятью, файловый ввод-вывод и вызовы безопасности не являются чем-то новым. У других операционных систем также есть подобные вызовы, хотя, как правило, их количество исчисляется не сотнями, как в Win32. Но что действительно отличает интерфейс Win32 — это тысячи и тысячи вызовов для графического интерфейса пользователя. Есть вызовы для создания, удаления, управления и использования окон, меню, инструментальных панелей, строк текущего состояния, линеек прокрутки, диалоговых окон, значков и многих других объектов, появляющихся на экране. Существуют вызовы для рисования геометрических фигур, заполнения их, управления использующимися ими цветовыми палитрами, управления шрифтами и для вывода графических изображений на экран. Наконец, есть вызовы для работы с клавиатурой, мышью и другими устройствами ввода, а также устройствами вывода, такими как звуковая карта, принтер и т. д. Короче говоря, интерфейс Win32 API (особенно его часть, относящаяся к графическому интерфейсу пользователя) огромен, и мы не смогли бы даже начать его подробное описание в данной главе, поэтому мы и пытаться не будем. Заинтересованные читатели могут обратиться к одной из многочисленных книг по Win32 (например, [265, 303, 271]).

Хотя интерфейс Win32 API также присутствует в системе Windows 98 (и в операционной системе для компактных мобильных компьютеров Windows CE), не во всех версиях Windows реализован каждый вызов, кроме того, иногда встречаются несущественные различия. Например, в системе Windows 98 нет средств безопасности, поэтому вызовы API, относящиеся к ней, просто возвращают в этой системе код ошибки. Для имен файлов в Windows 2000 используется кодировка Unicode, недоступная в Windows 98, а в именах файлов системы Windows 98, в отличие от имен файлов Windows 2000, не различаются строчные и прописные символы (хотя некоторые функции поиска файлов по имени не чувствительны к регистру). Кроме того, у некоторых вызовов в различных версиях операционной системы Windows могут различаться входные и выходные параметры. Например, в системе Windows 2000 все экранные координаты, задаваемые в качестве параметров графическим функциям, представляют собой действительно 32-разрядные числа, тогда как в Windows 98 используются только младшие 16 разрядов, так как большая часть графической подсистемы все еще остается 16-разрядной. Существование интерфейса Win32 API на нескольких различных операционных систем облегчает перенос программ с одной системы на другую, но благодаря наличию этих небольших различий требуется определенная аккуратность, чтобы программа оставалась переносимой.

Реестр

Операционной системе Windows приходится управлять большими объемами информации об оборудовании, программном обеспечении и пользователях. В Windows 3.x эта информация хранилась в сотнях файлов с расширением *.ini* (initialization — инициализация), разбросанных по всему диску. Начиная с Windows 95, почти вся информация, необходимая для загрузки и конфигурирования системы и настройки ее под конкретного пользователя, была собрана в одной большой центральной базе данных, называемой **реестром**. В данном разделе будет описан реестр Windows 2000.

Для начала отметим, что хотя многие части системы Windows 2000 сложны и запутанны, реестр является одним из самых запутанных мест в Windows, и похожие на шифр условные обозначения отнюдь не помогают в нем разобраться. К счастью, этой теме были посвящены целые книги [39, 154, 164]. Сама же идея реестра очень проста. Он состоит из набора каталогов, каждый из которых содержит либо подкаталоги, либо записи. В этом он похож на файловую систему, содержащую очень маленькие файлы.

Путаница начинается с того факта, что корпорация Microsoft называет каталог реестра **ключом**, чем он определенно не является. Более того, все каталоги верхнего уровня начинаются со строки *HKEY*, что означает «дескриптор ключа». У подкаталогов, как правило, лучшие имена, хотя и не всегда.

В нижней части этой иерархической структуры располагаются записи, называемые **значениями**, содержащие информацию. У каждого значения три части: имя, тип и данные. Имя представляет собой просто строку формата Unicode, часто `default`, если каталог содержит всего одно значение. Тип может быть одним из 11 стандартных типов. Среди наиболее часто используемых типов строка формата Unicode, 32-разрядное целое число, двоичное число произвольной длины и символьная ссылка на каталог или запись реестра. Символьные имена полностью аналогичны символьным ссылкам в файловых системах или значкам на рабочем столе Windows: они позволяют одной записи указывать на другую запись или каталог. Символьная ссылка также может использоваться как ключ, то есть то, что кажется каталогом, может на самом деле оказаться ссылкой на другой каталог.

На верхнем уровне в реестре Windows 2000 есть шесть ключей, называемых **корневыми ключами**, перечисленные в табл. 11.4. В этой таблице также показаны некоторые интересные **подключи** (подкаталоги). (Применение заглавных букв специального значения не имеет, но такова традиция корпорации Microsoft.) Увидеть этот список на своей системе можно с помощью одного из редакторов реестра, *regedit* или *regedit32*, которые, к сожалению, отображают различную информацию и используют различные форматы. С их помощью можно также изменять значения записей реестра. Начинающим пользователям не рекомендуется изменять какие-либо ключи или значения системного реестра, если они хотят впоследствии еще хотя бы раз загрузить эту систему. Однако просмотр реестра не грозит никакими последствиями. Итак, я вас предупредил.

Таблица 11.4. Корневые ключи и некоторые подключи реестра

Ключ	Описание
HKEY_LOCAL_MACHINE	Свойства аппаратного и программного обеспечения
HARDWARE	Описание аппаратуры и отображение аппаратуры на драйверы
SAM	Информация об учетных записях и правах доступа пользователей
SECURITY	Политика безопасности для всей системы
SOFTWARE	Общая информация об установленных прикладных программах
SYSTEM	Информация для загрузки системы
HKEY_USERS	
USER-AST-ID	Информация о пользователях; по одному ключу на пользователя
AppEvents	Профиль пользователя AST
Console	Какой звук, когда издавать (входящая электронная почта/факс, ошибка и т. д.)
Control	Установки командного режима (цвета, шрифты, история и т. д.)
Panel	Внешний вид рабочего стола, заставка, чувствительность мыши и т. д.
Environment	Переменные окружения
Keyboard Layout	Раскладка клавиатуры: 102-key US, AZERTY, Dvorak и т. д.
Printers	Информация об установленных принтерах
Software	Настройки пользователя для программного обеспечения корпорации Microsoft и других производителей
HKEY_PERFORMANCE_DATA	Сотни счетчиков, следящих за производительностью системы
HKEY_CLASSES_ROOT	Ссылка на HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
HKEY_CURRENT_CONFIG	Ссылка на текущий профиль аппаратного обеспечения
HKEY_CURRENT_USER	Ссылка на текущий профиль пользователя

Первый ключ (то есть каталог), HKEY_LOCAL_MACHINE, является, вероятно, наиболее важным, так как в нем содержится вся информация о локальной системе. У этого ключа есть пять подключей (подкаталогов). Подключ HARDWARE содержит множество подключей, в которых хранится вся информация об аппаратном обеспечении, например, какой драйвер какой частью аппаратуры управляет. Эта информация формируется на лету менеджером устройств plug-and-play во время загрузки системы. В отличие от других подключей этот подключ не хранится на диске.

Подключ SAM (Security Account Manager — администратор учетных данных в системе безопасности) хранит имена пользователей, групп, пароли, а также другую информацию об учетных записях пользователей, необходимую для регистрации в системе. Подключ SECURITY содержит данные об общей политике безопасности, например минимальную длину паролей, допустимое количество неудачных попыток регистрации и т. д.

Подключ SOFTWARE — это то место, в котором производители программного обеспечения хранят настройки программ. Если у пользователя в системе установлены программы *Acrobat*, *Photoshop* и *Premiere* компании Adobe, там будет содержаться подключ Adobe, под которым будут располагаться подключи для хранения Acrobat, Photoshop, Premiere и других программных продуктов компании Adobe. Записи в этих подкаталогах могут хранить все, что программистам компании Adobe понадобится поместить туда, — это номера версии и сборки, а также параметры уста-

новки пакета, сведения о драйверах и т. д. Наличие системного реестра избавляет их от хлопот по выдумыванию собственного метода хранения подобной информации. Специфическая для отдельных пользователей информация также хранится в реестре, но в ключе HKEY_USERS.

Подключ SYSTEM содержит главным образом информацию о загрузке системы, например список драйверов, которые требуется загрузить. Здесь также хранится список служб (демонов), которые должны быть запущены после загрузки, и сведения об их конфигурации.

Ключ верхнего уровня HKEY_USERS содержит профили для каждого пользователя. Все выбираемые пользователем настройки и параметры хранятся здесь. Когда пользователь изменяет какой-либо параметр при помощи панели управления, скажем, цветовую схему рабочего стола, новые установки записываются сюда. Многие программы в панели управления главным образом занимаются сбором информации, получая ее от пользователя, и сохраняют полученные сведения в реестре. Некоторые из подключей ключа HKEY_USERS показаны в табл. 11.4 и практически не требуют дополнительных комментариев. Другие подключи, например Software, содержат удивительно большое количество подключей, даже если в системе не установлено никаких пакетов программного обеспечения.

Ключ верхнего уровня HKEY_PERFORMANCE_DATA не содержит ни данных, считываемых с диска, ни данных, собираемых менеджером plug-and-play. Вместо этого данный ключ предоставляет окно в операционную систему. Сама система содержит сотни счетчиков для мониторинга производительности системы. К таким счетчикам можно получить доступ через этот ключ реестра. При обращении к подключу запускается специальная процедура, собирающая и возвращающая информацию (возможно, считывающая один или несколько счетчиков и объединяющая их определенным способом). В редакторах *regedit* или *regedit32* этот ключ не виден. Вместо редактора нужно воспользоваться одной из утилит измерения производительности, таких как *pfmon*, *perfmon* и *pview*. Существует множество подобных программ, они либо поставляются на компакт-диске Windows 2000, либо входят в пакет Resource Kit, либо производятся другими компаниями.

Остальных трех ключей верхнего уровня на самом деле не существует. Каждый из них представляет собой символьную ссылку на определенный подключ реестра. Самым интересным является ключ HKEY_CLASSES_ROOT. Он указывает на каталог, управляющий объектами COM (Component Object Model — модель компонентных объектов), а также занимающийся установкой соответствий между расширениями файлов и программами. Когда пользователь дважды щелкает мышью на файле, имя которого оканчивается на, скажем, *.doc*, программа, перехватывающая щелчок мыши, смотрит в это место реестра, чтобы определить, которую программу следует запустить (вероятно, *Microsoft Word*). Ключ HKEY_CLASSES_ROOT содержит целую базу данных распознаваемых расширений и соответствующих им программ.

Ключ HKEY_CURRENT_CONFIG представляет собой ссылку на подключ, содержащий информацию о текущей конфигурации аппаратного обеспечения. Пользователь может сформировать несколько конфигураций аппаратуры, например, отключая различные устройства, чтобы проверить, не они ли служили причиной странного поведения системы. Этот ключ указывает на текущую конфигурацию. Ключ HKEY_CURRENT_USER указывает на настройки текущего пользователя, что позволяет быстро находить их.

Ни один из последних трех ключей в действительности ничего не добавляет, так как эта информация уже была доступна (хотя и не в столь удобном виде). Таким образом, хотя редакторы *regedit* и *regedit32* перечисляют пять ключей верхнего уровня, на самом деле существуют только три каталога верхнего уровня, один из которых не отображается.

Реестр полностью доступен программисту Win32. Существуют вызовы для создания и удаления ключей, просмотра значений ключей и т. д. Некоторые наиболее важные вызовы перечислены в табл. 11.5.

Таблица 11.5. Некоторые вызовы Win32 API для работы с реестром

Функция Win32 API	Описание
RegCreateKeyEx	Создать новый ключ реестра
RegDeleteKey	Удалить ключ реестра
RegOpenKeyEx	Открыть ключ и получить его дескриптор
RegEnumKeyEx	Перенумеровать подключи, подчиненные ключу дескриптора
RegQueryValueEx	Искать данные по значению в ключе

Когда система выключается, большая часть данных реестра (но не вся, как уже упоминалось выше) сохраняется на диске в файлах, называемых **ульями**. Большинство этих файлов располагается в каталоге *\winnt\system32\config*. Поскольку их целостность представляет особую важность для правильной работы системы, при их обновлении автоматически создаются резервные копии, а запись выполняется при помощи атомарных транзакций, чтобы предотвратить порчу данных в случае сбоя системы во время записи. При потере реестра потребуется переустанавливать все программное обеспечение.

Структура системы

В предыдущих разделах мы рассмотрели Windows 2000 с точки зрения программиста. Теперь мы откроем капот и посмотрим, как устроена система внутри, что делают ее различные компоненты, как они взаимодействуют друг с другом и с программами пользователя. Хотя использованию операционной системы Windows 2000 посвящено множество книг, далеко не все они рассказывают о том, как работает эта система. Без сомнения, лучшим источником, в котором можно получить дополнительную информацию по этому вопросу, является [309]. В основе некоторой части материала данной главы лежит эта книга и сведения, полученные от ее авторов. Корпорация Microsoft также относится к основным источникам.

Структура операционной системы

Операционная система Windows 2000 состоит из двух основных частей: самой операционной системы, работающей в режиме ядра, и подсистем окружения, работающих в режиме пользователя. Ядро является традиционным ядром в том смысле, что оно управляет процессами, памятью, файловой системой и т. д. Подсистемы окружения представляют собой нечто необычное, так как они являются отдель-

ными процессами, помогающими пользователю выполнять определенные системные функции. В следующих разделах мы по очереди изучим обе части системы.

Одно из многих усовершенствований системы NT по сравнению с Windows 3.1 заключалось в ее модульной структуре. Она состояла из относительно небольшого ядра, работавшего в режиме ядра, плюс нескольких серверных процессов, работавших в режиме пользователя. Процессы пользователя взаимодействовали с серверными процессами с помощью модели клиент-сервер: клиент посылал серверу сообщение, а сервер выполнял определенную работу и возвращал клиенту результат в ответном сообщении. Такая модульная структура упрощала перенос системы на другие компьютеры. В результате операционная система Windows NT была успешно перенесена на платформы с процессорами, отличными от процессоров Intel, а именно: Alpha корпорации DEC, Power PC корпорации IBM и MIPS фирмы SGI. Кроме того, такая структура защищала ядро от ошибок в коде серверов. Однако для увеличения производительности, начиная с версии NT 4.0, довольно большая часть операционной системы (например, управление системными вызовами и вся экранная графика) были возвращены в ядро. Такая схема сохранилась и в Windows 2000.

Тем не менее в операционной системе Windows 2000 сохранилась некоторая структура. Система разделена на несколько уровней, каждый из которых пользуется службами лежащего ниже уровня. Эта структура проиллюстрирована на рис. 11.2. (Затененная область обозначает исполняющую систему. Квадратики, помеченные символом «D», обозначают драйверы устройств. Сервисные процессы являются системными демонами.) Один из уровней разделен горизонтально на множество модулей. У каждого модуля есть определенная функция, а также четко определенный интерфейс для взаимодействия с другими модулями.

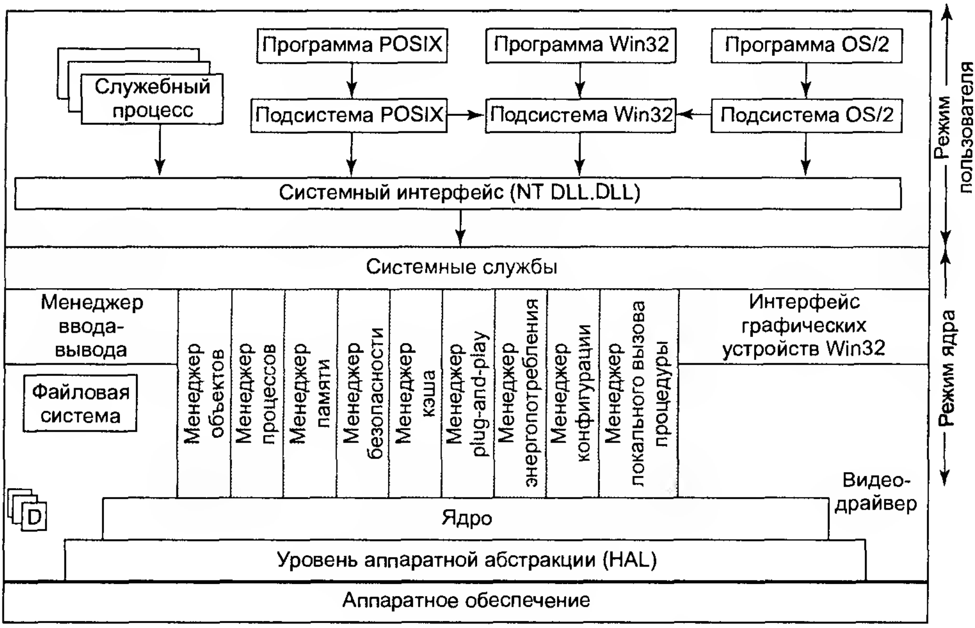


Рис. 11.2. Структура Windows 2000 (слегка упрощенная)

Два нижних уровня программного обеспечения, уровень аппаратных абстракций (HAL, Hardware Abstraction Layer) и ядро написаны на языке С и ассемблере и являются частично машинно-зависимыми. Верхние уровни написаны исключительно на С и почти полностью машинно-независимы. Драйверы написаны на С или, в некоторых случаях, на С++. В последующих разделах мы изучим различные компоненты системы, начиная с самых нижних уровней и постепенно продвигаясь наверх.

Уровень аппаратных абстракций

Одна из целей создания Windows 2000 (и Windows NT) заключалась в возможности переносить систему на другие платформы. В идеале при появлении новой машины для запуска операционной системы на ней нужно всего лишь перекомпилировать операционную систему новым компилятором для данной машины. К сожалению, жизнь не так легка. Хотя можно добиться полной переносимости верхних уровней операционной системы (так как в основном они имеют дело с внутренними структурами данных), нижние уровни работают с регистрами устройств, прерываниями, DMA и другими аппаратными особенностями, которые очень сильно отличаются на разных машинах. Хотя большая часть кода нижнего уровня написана на С, даже ее нельзя просто перенести с процессора Pentium на процессор Alpha, перекомпилировать и перезагрузить, так как существует большое количество мелких различий между этими процессорами, не имеющих отношения к различиям в наборе команд, которые невозможно спрятать компилятором.

Ясно представляя себе эту проблему, корпорация Microsoft предприняла серьезные попытки скрыть многие из аппаратных различий в тонком уровне на самом дне системы, названном **уровнем аппаратных абстракций** (**HAL**, Hardware Abstraction Layer). (Без всякого сомнения, имя HAL было позаимствовано у компьютера HAL в фильме Стэнли Кубрика *«2001: Космическая Одиссея»*. По слухам, Кубрик получил название своего компьютера из имени доминирующей в то время компьютерной корпорации IBM, в результате вычитания единицы из каждой буквы.)

Работа уровня HAL заключается в том, чтобы предоставлять всей остальной системе абстрактные аппаратные устройства, свободные от бородавок и индивидуальных отличительных особенностей, которыми так богато реальное аппаратное обеспечение. Эти устройства представляются в виде машинно-независимых служб (процедурных вызовов и макросов), которые могут использоваться остальной операционной системой и драйверами. Поскольку драйверы и ядро пользуются службами HAL (идентичными на всех операционных системах Windows 2000, независимо от аппаратного обеспечения) и не обращаются напрямую к устройствам, требуется значительно меньше изменений для их переноса на другую платформу. Перенос самого уровня HAL довольно прост, так как весь машинно-зависимый код сконцентрирован в одном месте, а цель переделки четко определена, то есть заключается в реализации всех служб уровня HAL.

В уровень HAL включены те службы, которые зависят от набора микросхем материнской платы и меняются от машины к машине в разумных предсказуемых пределах. Другими словами, он разработан, чтобы скрывать различия между материнскими платами различных производителей, но не различия между процессорами Pentium и Alpha. К службам уровня HAL относятся: доступ к регистрам

устройств, адресация к устройствам, независящим от шины, обработка прерываний и возврат из прерываний, операции DMA (Direct Memory Access — прямой доступ к памяти), управление таймерами, часами реального времени, спин-блокировками нижнего уровня и синхронизация многопроцессорных конфигураций, интерфейс с BIOS и доступ к CMOS-памяти. Уровень HAL не предоставляет абстракций или служб для специфических устройств ввода-вывода — клавиатур, мышей или дисков, а также блоков управления памятью MMU.

В качестве примера того, что делает уровень аппаратных абстракций, рассмотрим вопрос устройств ввода-вывода с отображаемыми на память регистрами и устройств ввода-вывода, доступ к которым осуществляется через порты. На некоторых машинах используется один способ доступа к устройствам ввода-вывода, а на других машинах — другой. Как должен быть запрограммирован драйвер: на использование портов или регистров? Вместо того чтобы заставлять делать выбор в пользу одного или другого метода (что приведет к невозможности переноса драйвера с одной платформы на другую), уровень аппаратных абстракций предоставляет три процедуры для чтения регистров устройств и еще три для записи в них:

```
uc = READ_PORT_UCHAR(port);      WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);     WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);      WRITE_PORT_LONG(port, ul);
```

Эти процедуры читают и пишут соответственно 8-, 16- и 32-разрядные целые без знака числа в указанный порт. Реализацией этих действий в виде обращения к физическим портам или регистрам, отображаемым на память, занимается уровень HAL. Таким образом, драйвер без каких-либо изменений может быть перемещен на совершенно иную платформу.

Драйверам часто бывает нужно получить доступ к специфическим устройствам ввода-вывода. На аппаратном уровне у драйвера есть один или несколько адресов определенной шины. Поскольку у современных компьютеров часто есть несколько шин (ISA, PCI, SCSI, USB, 1394 и т. д.), может случиться, что два или более устройств имеют один и тот же адрес шины, поэтому требуется некоторый способ отличать эти устройства. Уровень HAL предоставляет службу для идентификации устройств, отображая адреса устройств на шине на логические системные адреса. Поэтому драйверам не нужно следить за тем, которое устройство находится на какой шине. Такая логическая адресация аналогична дескрипторам, выдаваемым операционной системой программам пользователя для обращения к файлам и другим системным ресурсам. Этот механизм также защищает более высокие уровни от свойств структур шин и соглашений об адресации.

С прерываниями связана схожая проблема — они также являются зависимыми от шины. Здесь уровень HAL предоставляет службы для именования прерываний уникальным в пределах всей системы способом, а также службы, позволяющие драйверам связывать процедуры обработки прерываний с прерываниями переносимым способом. При этом не нужно знать, какой вектор к какой шине относится. Управление уровнем запроса прерывания также осуществляется на уровне HAL.

Другая служба HAL занимается управлением операциями DMA независимым от устройств способом. HAL может управлять как единым для всей системы механизмом DMA, так и механизмами DMA, специфичными для конкретных плат ввода-вывода. Обращение к устройствам осуществляется по их логическим адресам.

Уровень HAL также реализует программные операции чтения/записи с разнесением данных (с обращением к не являющимся соседними блокам памяти).

Уровень HAL управляет часами и таймерами, обеспечивая переносимость работающих с ними программ. Время хранится в интервалах по 100 нс, начиная с 1 января 1601 года, что существенно точнее, чем то, как это делалось в MS-DOS (в 2-секундных интервалах с 1 января 1980 года). К тому же новый способ хранения времени позволяет учитывать относящуюся к компьютерам деятельность в XVII–XIX веках. Временные службы уровня HAL обеспечивают независимость драйверов от фактических частот, на которых работают часы.

Иногда требуется синхронизация компонентов ядра на очень низком уровне, особенно для того, чтобы избежать конфликтов на многопроцессорных системах. Уровень HAL предоставляет несколько примитивов для управления этой синхронизацией. Примером являются спин-блокировки, в которых один центральный процессор просто ждет, пока другой центральный процессор не освободит определенный ресурс. В частности, такой метод синхронизации применяется в ситуациях, в которых доступ к ресурсу, как правило, получается всего на несколько команд процессора.

Наконец, после загрузки операционной системы уровень HAL общается с BIOS и инспектирует память конфигурации CMOS, если она используется, чтобы определить, какие шины и устройства ввода-вывода содержатся в системе и как их следует настроить. Затем эта информация помещается в реестр, чтобы другие компоненты системы могли просматривать их, не обращаясь напрямую к BIOS или CMOS-памяти. Схематично набор функций, выполняемый уровнем HAL, показан на рис. 11.3.

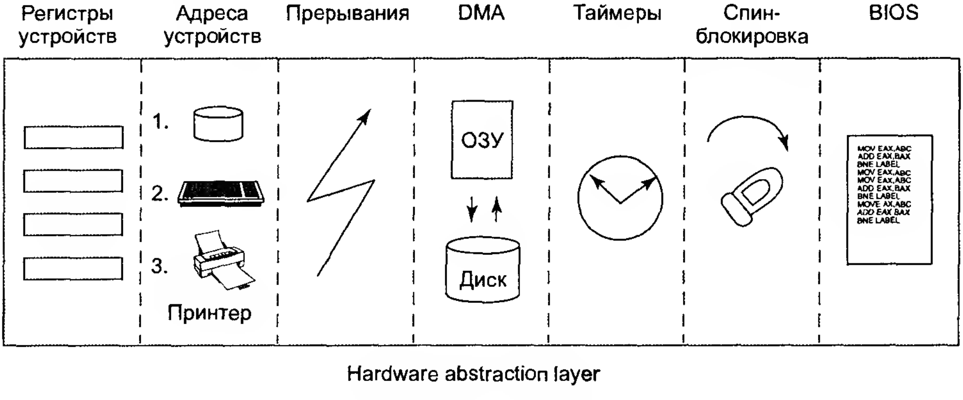


Рис. 11.3. Некоторые функции уровня HAL

Поскольку уровень HAL является в большой степени машинно-зависимым, он должен в совершенстве соответствовать системе, на которой установлен, поэтому набор различных уровней HAL поставляется на компакт-диске Windows 2000. Во время установки системы из них выбирается подходящий уровень и копируется на жесткий диск в системный каталог *\winnt\system32* в виде файла *hal.dll*. При всех последующих загрузках операционной системы используется эта версия уровня HAL. Если удалить этот файл, то система загрузиться не сможет.

Хотя эффективность уровня HAL является довольно высокой, для мультимедийных приложений ее может быть недостаточно. По этой причине корпорация Microsoft также производит пакет программного обеспечения, называемый **DirectX**, расширяющий функциональность уровня HAL дополнительными процедурами и предоставляющий пользовательским процессам прямой доступ к аппаратному обеспечению. Пакет DirectX является специализированным, поэтому мы не станем обсуждать его в дальнейшем в данной главе.

Уровень ядра

Над уровнем аппаратных абстракций располагается уровень, содержащий то, что корпорация Microsoft называет **ядром**, а также драйверы устройств. В некоторых старых документах ядро называлось «микроядром», которым оно никогда не было, так как менеджер памяти, файловая система и другие основные компоненты системы постоянно находились в пространстве ядра и с самого начала работали в режиме ядра. Ядро определенно не является микроядром и сейчас, так как, начиная с NT 4.0, практически вся операционная система была помещена в пространство ядра.

В главе, посвященной операционной системе UNIX, мы использовали термин «ядро» для обозначения всего, что работает в режиме ядра. В данной главе мы (весьма неохотно) зарезервируем этот термин для обозначения части системы на рис. 11.2, помеченной этим словом, а все программное обеспечение, работающее в режиме ядра, станем называть «операционной системой». Часть ядра (и большая часть уровня HAL) постоянно находится в оперативной памяти (то есть не выгружается). При помощи установки соответствующего приоритета эта часть ядра может решать, допустимо ли прерывание от устройств ввода-вывода или нет. Хотя значительная часть ядра представляет собой машинно-зависимую программу, тем не менее большая ее часть написана на С, кроме тех мест, в которых производительность считается важнее всех остальных задач.

Назначение ядра заключается в том, чтобы сделать всю остальную часть операционной системы независимой от аппаратуры и, таким образом, легко переносимой на другие платформы. Оно начинается там, где заканчивается уровень HAL. Ядро получает доступ к аппаратуре через уровень HAL. Оно построено на чрезвычайно низкоуровневых службах уровня HAL, формируя из них абстракции более высоких уровней. Например, у уровня HAL есть вызовы для связывания процедур обработки прерываний с прерываниями и установки их приоритетов, но больше практически ничего уровень HAL в этой области не делает. Ядро, напротив, предоставляет полный механизм для переключения контекста. Оно должным образом сохраняет все регистры центрального процессора, изменяет таблицы страниц, сохраняет кэш центрального процессора и т. д. Когда все эти действия выполнены, работавший ранее поток оказывается полностью сохраненным в таблицах, расположенных в памяти. Затем ядро настраивает карту памяти нового потока и загружает его регистры, после чего новый поток готов к работе.

Программа планирования потоков также располагается в ядре. Когда наступает пора проверить, не готов ли к работе новый поток, например, после того, как истечет выделенный потоку квант времени или по завершении процедуры обработки прерываний ввода-вывода, ядро выбирает поток и выполняет переключение контекста, необходимое, чтобы запустить этот поток. С точки зрения остальной

операционной системы, переключение потоков автоматически осуществляется более низкими уровнями, так что для более высоких уровней не остается никакой работы. Сам алгоритм планирования будет обсуждаться ниже в этой главе, когда мы подойдем к теме процессов и потоков.

Помимо предоставления абстрактной модели аппаратуры более высоким уровням и управления переключениями потоков, ядро также выполняет еще одну ключевую функцию: предоставляет низкоуровневую поддержку двум классам объектов — управляющим объектам и объектам диспетчеризации. Эти объекты не являются объектами, к которым пользовательские процессы получают дескрипторы, но представляют собой внутренние объекты, на основе которых исполняющая система строит объекты пользователя.

Управляющие объекты — это объекты, управляющие системой, включая примитивные объекты процессов, объекты прерываний и два несколько странных объекта, называемых DPC и APC. Объект **DPC** (Deferred Procedure Call — отложенный вызов процедуры) используется, чтобы отделить часть процедуры обработки прерываний, для которой время является критичным, от той ее части, для которой время некритично. Как правило, процедура обработки прерываний сохраняет несколько аппаратных регистров, связанных с прерывающим устройством ввода-вывода, чтобы их можно было потом восстановить, и разрешает аппаратуре продолжать работу, но оставляет большую часть обработки на потом.

Например, когда пользователь нажимает на клавишу, процедура обработки прерываний от клавиатуры считывает из регистра код нажатой клавиши и разрешает прерывания от клавиатуры. Но эта процедура не должна немедленно обрабатывать введенный символ, особенно если в данный момент происходит нечто более важное (то есть нечто с более высоким приоритетом). Пока обработка клавиши занимает не более 100 мс, пользователь ничего не заметит. Отложенные вызовы процедуры также применяются для слежения за таймерами и другой активностью, для которой не требуется немедленная обработка. Очередь DPC представляет собой механизм напоминания о том, что есть работа, которую следует выполнить позднее.

Объект **APC** (Asynchronous Procedure Call — асинхронный вызов процедуры) похож на отложенный вызов процедуры DPC, но отличается тем, что асинхронный вызов процедуры выполняется в контексте определенного процесса. Когда обрабатывается нажатая клавиша, не имеет значения, в каком контексте работает DPC, так как все, что требуется сделать, — это исследовать введенный код и, возможно, поместить его в буфер в ядре. Однако если по прерыванию потребуется скопировать буфер из пространства ядра в адресное пространство пользовательского процесса (например, по завершении операции чтения модема), тогда процедура копирования должна работать в контексте получателя. Контекст получателя нужен для того, чтобы в таблице страниц одновременно содержались и буфер ядра, и буфер пользователя (все процессы, как мы увидим в дальнейшем, содержат в своем адресном пространстве все ядро целиком). По этой причине в разных ситуациях используются APC или DPC.

Еще один тип объектов ядра — **объекты диспетчеризации**. К ним относятся семафоры, мьютексы, события, таймеры и другие объекты, изменения состояния которых могут ждать потоки. Причина, по которой они должны (частично) обрабатываться ядром, заключается в том, что они тесно переплетены с планированием

потоками, что входит в круг задач ядра. Кстати, мьютексы в программах назывались «мутантами», так как от них требовалось соблюдение семантики операционной системы OS/2. То есть они не должны были автоматически разблокироваться, когда процесс, захвативший их, прекращал свою работу. Такое поведение разработчики Windows 2000 посчитали странным. (Семантика OS/2 использовалась, потому что изначально предполагалось, что NT заменит эту операционную систему, поставлявшуюся с компьютерами PC/2 корпорации IBM.)

Исполняющая система

Над ядром и драйверами устройств располагается верхняя часть операционной системы, называемая **исполняющей системой** (а также иногда супервизором или диспетчером), показанная в виде затененной области на рис. 11.2. Исполняющая система написана на С, она не зависит от архитектуры и может быть перенесена на новые машины с относительно небольшими усилиями. Исполняющая система состоит из 10 компонентов, каждый из которых представляет собой просто набор процедур, работающих вместе для выполнения некоторой задачи. Между отдельными компонентами нет жестких границ, и различные авторы, описывающие исполняющую систему, могут даже по-разному группировать составляющие ее процедуры в компоненты. Следует заметить, что компоненты одного уровня могут вызывать друг друга, и на практике они этим довольно активно занимаются.

Менеджер объектов управляет всеми объектами, известными операционной системе. К ним относятся процессы, потоки, файлы, каталоги, семафоры, устройства ввода-вывода, таймеры и многое другое. При создании объекта менеджер объектов получает в адресном пространстве ядра блок виртуальной памяти и возвращает этот блок в список свободных блоков, когда объект уничтожается. Его работа заключается в том, чтобы следить за всеми объектами.

Чтобы избежать путаницы, отметим, что большинство компонентов исполняющей системы, помеченные на рис. 11.2 как «менеджеры», не являются процессами или потоками, а представляют собой просто набор процедур, которые могут выполняться другими потоками в режиме ядра. Однако некоторые из них, такие как менеджер питания и менеджер plug-and-play, являются настоящими потоками.

Менеджер объектов также управляет пространством имен, в которое помещается созданный объект, чтобы впоследствии к нему можно было обратиться по имени. Все остальные компоненты исполняющей системы активно пользуются объектами во время своей работы. Объекты занимают центральное место в функционировании операционной системы Windows 2000, поэтому они будут подробно обсуждаться в следующем разделе.

Менеджер ввода-вывода формирует каркас для управления устройствами ввода-вывода и предоставляет общие службы ввода-вывода. Он предоставляет остальной части системы независимый от устройств ввод-вывод, вызывая для выполнения физического ввода-вывода соответствующий драйвер. Здесь также располагаются все драйверы устройств (обозначены символом «D» на рис. 11.2). Файловые системы формально являются драйверами устройств под управлением менеджера ввода-вывода. Существует два драйвера для файловых систем FAT и NTFS, независимые друг от друга и управляющие различными разделами диска. Все файловые системы FAT управляются одним драйвером. Ввод-вывод мы рас-

смотрим в разделе «Ввод-вывод в Windows 2000», а одну из файловых систем, NTFS — в разделе «Файловая система Windows 2000».

Менеджер процессов управляет процессами и потоками, включая их создание и завершение. Он занимается не стратегиями, применяемыми по отношению к процессам, а механизмом, используемым для управления ими. Менеджер процессов основывается на объектах потоков и процессов ядра и добавляет к ним дополнительные функции. Это ключевой элемент многозадачности в Windows 2000. Управление процессами будет рассматриваться в разделе «Процессы и потоки в Windows 2000».

Менеджер памяти реализует архитектуру виртуальной памяти со страничной подкачкой по требованию операционной системы Windows 2000. Он управляет преобразованием виртуальных страниц в физические страничные блоки. Таким образом, он реализует правила защиты, ограничивающие доступ каждого процесса только теми страницами, которые принадлежат его адресному пространству, а не адресным пространствам других процессов (кроме специальных случаев). Он также контролирует определенные системные вызовы, относящиеся к виртуальной памяти. Управление памятью будет рассматриваться в разделе «Управление памятью».

Менеджер безопасности приводит в исполнение сложный механизм безопасности Windows 2000, удовлетворяющий требованиям класса C2 Оранжевой книги Министерства обороны США. В Оранжевой книге перечислено множество правил, которые должна соблюдать система, начиная с аутентификации при регистрации и заканчивая управлением доступом, а также обнулением страниц перед их повторным использованием. Менеджер безопасности будет обсуждаться в разделе «Безопасность в Windows 2000».

Менеджер кэша хранит в памяти блоки диска, которые использовались в последнее время, чтобы ускорить доступ к ним в случае, если они понадобятся вновь. Его работа состоит в том, чтобы определить, какие блоки понадобятся снова, а какие нет. Операционная система Windows 2000 может одновременно использовать несколько файловых систем. В этом случае менеджер кэша обслуживает все файловые системы, таким образом, каждой файловой системе не нужно заниматься управлением собственного кэша. Когда требуется блок, он запрашивается у менеджера кэша. Если у менеджера кэша нет блока, он обращается за блоком к соответствующей файловой системе. Поскольку файлы могут отображаться в адресное пространство процессов, менеджер кэша должен взаимодействовать с менеджером виртуальной памяти, чтобы обеспечить требуемую непротиворечивость. Количество памяти, выделенной для кэша, динамически изменяется и может увеличиваться или уменьшаться при необходимости. Менеджер кэша будет описан в разделе «Кэширование в Windows 2000».

Менеджер plug-and-play получает все уведомления об установленных новых устройствах. Для некоторых устройств проверка производится при загрузке системы, но не после нее. Другие устройства, например устройства USB (Universal Serial Bus — универсальная последовательная шина), могут подключаться в любое время, и их подключение запускает пересылку сообщения менеджеру plug-and-play, который затем находит и загружает соответствующий драйвер.

Менеджер энергопотребления управляет потреблением электроэнергии. Он выключает монитор и диски, если к ним не было обращений в течение определенного интервала времени. На переносных компьютерах менеджер энергопотребления следит за состоянием батарей и, когда заряд батарей подходит к концу, предпринимает соответствующие действия. Эти действия, как правило, заключаются в том, что он сообщает работающим программам о состоянии батарей. В результате программы могут сохранить свои файлы и приготовиться к корректному завершению работы.

Менеджер конфигурации отвечает за состояние реестра. Он добавляет новые записи и ищет запрашиваемые ключи.

Менеджер вызова локальной процедуры обеспечивает высокоэффективное взаимодействие между процессами и их подсистемами. Поскольку этот путь нужен для выполнения некоторых системных вызовов, эффективность оказывается критичной, вот почему для этого не используются стандартные механизмы межпроцессного взаимодействия.

Исполняющий модуль Win32 GDI обрабатывает определенные системные вызовы (но не все). Изначально он располагался в пространстве пользователя, но в версии NT 4.0 для увеличения производительности был перенесен в пространство ядра. **Интерфейс графических устройств GDI** (Graphic Device Interface) занимается управлением графическими изображениями для монитора и принтеров. Он предоставляет системные вызовы, позволяющие пользовательским программам выводить данные на монитор и принтеры независящим от устройств способом. Он также содержит оконный менеджер и драйвер дисплея. До версии NT 4.0 интерфейс графических устройств также находился в пространстве пользователя, но производительность при этом оставляла желать лучшего, поэтому корпорация Microsoft переместила его в ядро. Следует отметить, что рис. 11.2 создан безо всякого соблюдения масштаба. Так, например, интерфейс Win32 и модуль GDI, вместе взятые, превосходят всю остальную исполняющую систему.

Над исполняющей системой размещается тонкий уровень, называемый **системными службами**. Его функции заключаются в предоставлении интерфейса к исполняющей системе. Он принимает настоящие системные вызовы Windows 2000 и вызывает другие части исполняющей системы для их выполнения.

При загрузке операционная система Windows 2000 загружается в память как набор файлов. Основная часть операционной системы, состоящая из ядра и исполняющей системы, хранится в файле *ntoskrnl.exe.* Уровень HAL представляет собой библиотеку общего доступа, расположенную в отдельном файле *hal.dll.* Интерфейс Win32 и интерфейс графических устройств хранятся вместе в третьем файле, *win32k.sys.* Наконец, загружается множество драйверов устройств. У большинства из них расширение *.sys.*

На самом деле все не совсем так просто. Существует две версии файла *ntoskrnl.exe:* для однопроцессорных и многопроцессорных систем. Кроме того, существуют версии для процессора Xeon, способного поддерживать более 4 Гбайт физической памяти, и для процессора Pentium, который так много оперативной памяти поддержать не может. Наконец, этот модуль может содержать или не содержать

отладочные функции, в зависимости от чего он предназначается либо для отладки системы, либо для продажи в магазинах. Всего получается восемь комбинаций, но две пары были объединены вместе, и в результате остается только шесть. Одна из них копируется при установке в файл *ntoskrnl.exe*.

Следует сказать несколько слов об отладочных версиях. Когда на персональный компьютер устанавливается новое устройство ввода-вывода, то требуется установка драйвера, поставляемого производителем устройства, чтобы оно могло работать. Предположим, что карта IEEE 1394 установлена на компьютере и вроде бы нормально работает. Спустя две недели система внезапно рушится. Кого будет винить в этом владелец компьютера? Конечно, корпорацию Microsoft.

Ошибка, действительно, может произойти по вине корпорации Microsoft, но на самом деле некоторые ошибки бывают в неверно написанных драйверах, которые корпорация Microsoft не может контролировать и которые устанавливаются в память ядра и получают полный доступ к таблицам ядра, а также и ко всему аппаратному обеспечению. Пытаясь уменьшить количество телефонных звонков от разгневанных покупателей, корпорация Microsoft старается помочь авторам драйверов отладить их программы при помощи внедрения в различные места этих программ операторов вида

```
ASSERT(некоторое условие)
```

Эти операторы проверяют некоторые условия (например, допустимость тех или иных параметров). В коммерческой версии операционной системы все операторы *ASSERT* определены как макрос, который ничего не выполняет, таким образом, проверка из системы удаляется. В отладочной версии они определены как

```
#define ASSERT(a) if (! (a)) error( ... ),
```

в результате чего все проверки после трансляции ядра системы оказываются в исполняемом коде файла *ntoskrnl.exe* и могут выполняться во время работы системы. Хотя наличие подобных проверок страшно замедляет работу системы, эти проверки помогают авторам отладить свои драйверы, прежде чем они будут переданы покупателям. В отладочные версии системы также включено множество других функций отладки.

Драйверы устройств

Последняя часть схемы на рис. 11.2 состоит из **драйверов устройств**. Каждый драйвер может управлять одним или несколькими устройствами ввода-вывода, но драйвер устройства может также выполнять действия, не относящиеся к какому-либо специфическому устройству — шифровать поток данных или даже просто предоставлять доступ к структурам данных ядра. Драйверы устройств не являются частью двоичного файла *ntoskrnl.exe*. Преимущество такого подхода заключается в том, что как только драйвер устанавливается в систему, он добавляется в реестр и затем динамически загружается при каждой загрузке системы. Таким образом, файл *ntoskrnl.exe* остается одинаковым для всех конфигураций систем, но каждая система точно настраивается на конфигурацию аппаратуры.

Существуют драйверы для реально видимых и осязаемых устройств ввода-вывода, таких как диски и принтеры, но также есть драйверы для многих внутренних

устройств и микросхем, о которых практически никто ничего не слышал. Кроме того, как уже было сказано, файловые системы также представлены в виде драйверов устройств. Самый большой драйвер устройства для интерфейса Win32 и GDI показан на правой стороне рис. 11.2. Он обрабатывает множество системных вызовов и управляет большей частью графики. Поскольку пользователи могут устанавливать новые драйверы, у них есть возможность изменить содержимое ядра и повредить систему. По этой причине драйверы следует писать с большой осторожностью.

Реализация объектов

Объекты представляют собой, вероятно, самое важное понятие операционной системы Windows 2000. Они предоставляют однородный и непротиворечивый интерфейс ко всем системным ресурсам и структурам данных, таким как процессы, потоки, семафоры и т. д. У этой однородности есть много граней. Во-первых, все объекты именуются по одной и той же схеме. Доступ ко всем объектам также предоставляется одинаково, при помощи дескрипторов объектов. Во-вторых, поскольку доступ к объектам всегда осуществляется через менеджер объектов, все проверки, связанные с защитой, могут быть размещены в одном месте, с гарантией, что ни один процесс не сможет обойти их. В-третьих, возможно совместное использование объектов по одной и той же схеме. В-четвертых, поскольку все объекты открываются и закрываются через менеджер объектов, несложно отследить, какие объекты все еще используются, а какие можно безопасно удалить. В-пятых, эта однородная модель для управления объектов позволяет легко регулировать квоты ресурсов.

Ключом к пониманию объектов является тот факт, что исполняемый объект представляет собой просто набор последовательных слов в памяти (то есть в виртуальном адресном пространстве ядра). Объект представляет собой структуру данных в памяти, не больше и не меньше. Файл на диске не является объектом, хотя для файла при его открытии создается объект (то есть структура данных в виртуальном адресном пространстве ядра). Из того факта, что объекты представляют собой всего лишь структуры данных в виртуальном адресном пространстве ядра, следует, что при перезагрузке (или сбое) системы все объекты теряются. Действительно, когда операционная система загружается, нет никаких объектов (кроме бездействующих системных процессов, чьи объекты жестко прошиты в файле *ntoskrnl.exe*). Все остальные объекты создаются на ходу при загрузке системы и во время работы различных программ инициализации, а позднее пользовательских программ.

Структура объектов показана на рис. 11.4. Каждый объект содержит заголовок с определенной информацией, общей для всех объектов всех типов. Поля заголовка включают имя объекта, каталог, в котором объект живет в пространстве объектов, информацию защиты (при открытии объекта выполняется определенная проверка), а также список процессов, у которых есть открытые дескрипторы к данному объекту (если установлен определенный флаг отладки).

Каждый заголовок объекта также содержит поле цены квоты, представляющей собой плату, взимаемую с процесса за открытие объекта. Если файловый объект стоит один пункт, а процесс принадлежит к заданию, у которого есть 10 пунктов квоты, то суммарно все процессы этого задания могут открыть не более 10 файлов. Таким образом, для объектов каждого типа могут реализовываться ограничения на ресурсы.

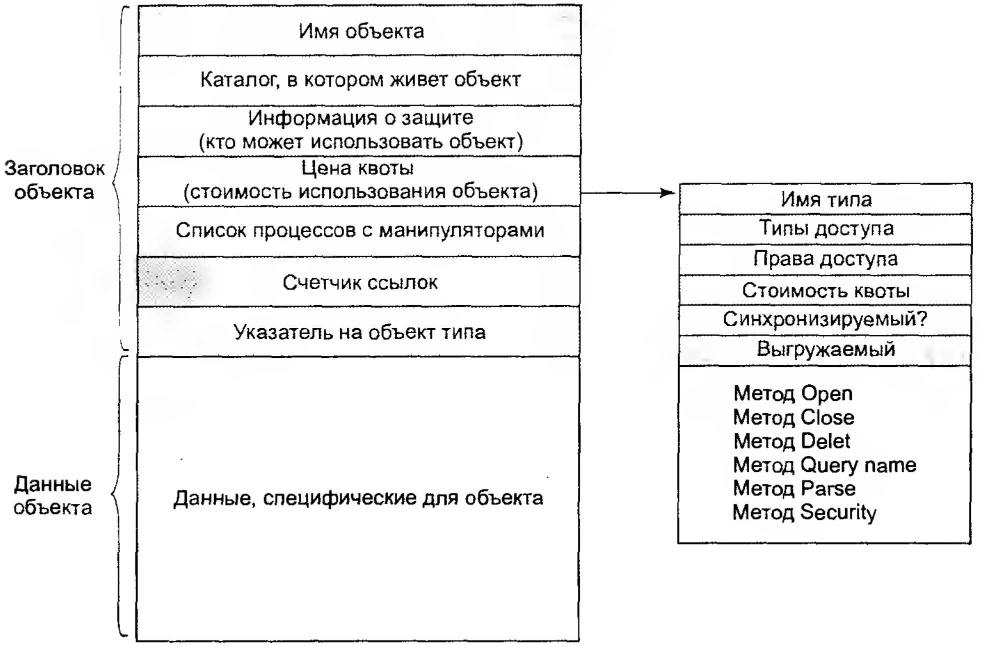


Рис. 11.4. Структура объекта

Объекты занимают важный ресурс — участки виртуального адресного пространства ядра — поэтому, когда объект более не нужен, он должен быть удален, а его адресное пространство возвращено системе. Для этого в заголовке каждого объекта содержится счетчик ссылок на объект. Этот счетчик увеличивается на единицу каждый раз, когда объект открывается, и уменьшается на единицу при закрытии объекта. Когда значение счетчика уменьшается до 0, это означает, что никто более не пользуется этим объектом. Когда объект открывается или освобождается компонентом исполняющей системы, используется второй счетчик, даже если настоящий дескриптор при этом не создается. Когда оба счетчика уменьшаются до 0, это означает, что этот объект более не используется ни одним пользователем и ни одним исполняющим процессом, то есть объект может быть удален, а его память освобождена.

Менеджеру объектов бывает необходимо получать доступ к динамическим структурам данных (объектам), но он не единственная часть исполняющей системы, которой это нужно. Другим частям исполняющей системы также бывает нужно динамически получать на время участки памяти. Для этого исполняющая система содержит два пула в адресном пространстве ядра: для объектов и для других

динамических структур данных. Эти пулы работают как «кучи», подобно процедурам языка C *malloc* и *free* для управления динамическими данными. Один пул является выгружаемым, а другой невыгружаемым (фиксированным в памяти). Объекты, обращения к которым часты, хранятся в невыгружаемом пуле; объекты, обращения к которым редки, например ключи реестра и некоторая информация, относящаяся к безопасности, хранятся в выгружаемом пуле. Когда памяти не хватает, этот пул может быть выгружен на диск и загружен обратно по страничному прерыванию. В действительности значительная часть программ и структур данных операционной системы также является выгружаемой, что позволяет снизить потребление памяти. Объекты, которые могут понадобиться, когда система выполняет критический участок программы (и когда подкачка не разрешается), должны храниться в невыгружаемом пуле. Когда требуется небольшое количество памяти, страница может быть получена из любого пула, а затем разбита на мелкие участки размером от 8 байт.

Объекты подразделяются на типы. Это означает, что у каждого объекта есть свойства, общие для всех объектов этого типа. Тип объекта определяется указателем на объект типа, как показано на рис. 11.4. Информация о типе объекта включает такие пункты, как название типа, данные о том, может ли поток ждать изменения состояния этого объекта (да для мьютексов, нет для открытых файлов), и должен ли объект этого типа храниться в выгружаемом или невыгружаемом пуле. Каждый объект указывает на свой объект типа.

Наконец, самая важная часть объекта — это указатели на программы для определенных стандартных операций, таких как `open`, `close` и `delete`. Когда вызывается одна из этих операций, используется указатель на типовой объект, в котором выбирается и выполняется соответствующая процедура. Такой механизм предоставляет системе возможность инициализировать новые объекты и освобождать память при их удалении.

Компоненты исполняющей системы могут динамически создавать новые типы. Фиксированного списка типов объектов не существует, но некоторые наиболее употребительные типы приведены в табл. 11.6. Давайте кратко рассмотрим эти типы объектов. С процессом и потоком все ясно. Существует один объект для каждого процесса и для каждого потока. В объекте хранятся основные свойства, необходимые для управления этим процессом или потоком. Следующие три объекта, семафор, мьютекс и событие, имеют отношение к синхронизации процессов. Семафоры и мьютексы работают так, как и ожидается, но с дополнительными функциями (например, максимальными значениями и тайм-аутами). События могут быть в одном из двух состояний: сигнализирующем и несигнализирующем. Если поток ждет события, находящегося в сигнализирующем состоянии, он немедленно получает управление. Если же ожидаемое потоком событие находится в несигнализирующем состоянии, тогда поток блокируется до тех пор, пока какой-либо другой поток не переведет это событие в сигнализирующее состояние (проще говоря, пока это событие не произойдет). Событие может также быть настроено таким образом, что после получения сигнала ожидающим его процессом это событие автоматически перейдет в несигнализирующее состояние. В противном случае событие останется в сигнализирующем состоянии.

Таблица 11.6. Некоторые общие типы объектов исполняющей системы, управляемых менеджером объектов

Тип	Описание
Процесс	Процесс пользователя
Поток	Поток внутри процесса
Семафор	Семафор со счетчиком, используемый для синхронизации процессов
Мьютекс	Двоичный семафор, используемый для входа в критическую область
Событие	Объект синхронизации с перманентным состоянием (сигнализирующий/нет)
Порт	Механизм для передачи сообщений между процессами
Таймер	Объект, позволяющий потоку спать в течение фиксированного интервала времени
Очередь	Объект, используемый для уведомления о завершении асинхронного ввода-вывода
Открытый файл	Объект, ассоциированный с открытым файлом
Маркер доступа	Описатель защиты для некоторого объекта
Профиль	Структура данных, используемая для анализа использования центрального процессора
Секция	Структура, используемая для отображения файлов на виртуальное адресное пространство
Ключ	Ключ реестра
Каталог объектов	Каталог для группирования объектов в менеджере объектов
Символьная ссылка	Указатель на другой объект по имени
Устройство	Объект устройства ввода-вывода
Драйвер устройства	У каждого загруженного драйвера устройства есть свой объект

Объекты порт, таймер и очередь также имеют отношение к связи и синхронизации. Порты представляют собой каналы между процессами, использующиеся для обмена сообщениями. Таймеры предоставляют способ блокировать процесс или поток на определенный срок. Очереди применяются для уведомления потоков о том, что начатая ранее асинхронная операция ввода-вывода завершена.

Объекты открытых файлов создаются при открытии файла. У неоткрытых файлов нет объектов, управляемых менеджером объектов. Маркеры доступа представляют собой объекты безопасности. Они идентифицируют пользователя и сообщают, какие привилегии имеет этот пользователь. Профили представляют собой структуры, используемые для хранения периодически фиксируемых значений счетчика команд работающего потока, которые позволяют определить, на что данная программа тратит свое время.

Секции являются объектами, используемыми системой памяти для управления отображаемыми на память файлами. Они хранят сведения о том, какой файл (или часть файла) на какие адреса памяти отображается. Ключи представляют собой ключи реестра и применяются для установки связи между именем и значением. Каталоги объектов являются полностью локальными по отношению к менеджеру объектов. Они предоставляют способ объединять связанные объекты тем же способом, каким обычные каталоги объединяют файлы в файловой системе. Символьные ссылки также подобны своим двойникам в файловой системе: они

позволяют имени в одной части пространства имен объектов ссылаться на объект в другой части этого пространства имен. У каждого известного системе устройства есть объект устройства, содержащий информацию о нем и использующийся для ссылки на устройство в системе. Наконец, у каждого загруженного драйвера устройства есть объект в пространстве объектов.

Пользователи могут создавать новые объекты или открывать уже существующие объекты при помощи вызовов Win32, таких как `CreateSemaphore` и `OpenSemaphore`. Эти вызовы являются библиотечными процедурами, которые в конечном итоге обращаются к настоящим системным вызовам. При успешном выполнении первый вызов создает, а второй открывает объект, создавая в результате 64-разрядную запись в таблице дескрипторов, хранящуюся в приватной таблице дескрипторов процесса в памяти ядра. Пользователю для последующей работы возвращается 32-разрядный индекс, указывающий положение дескриптора в таблице.

64-разрядный элемент таблицы дескрипторов в ядре содержит два 32-разрядных слова. Одно слово содержит 29-разрядный указатель на заголовок объекта. Младшие три разряда используются как флаги (например, указывающие, наследуется ли дескриптор дочерним процессом). Когда указатель используется, эти разряды маскируются. Второе слово содержит 32-разрядную маску прав доступа. Она нужна, потому что проверка разрешений выполняется только в то время, когда объект создается или открывается. Если у процесса есть только разрешение для чтения объекта, тогда все остальные биты маски будут нулями, что дает системе возможность отвергать любую операцию, кроме операции чтения.

На рис. 11.5 показаны таблицы дескрипторов для двух процессов и их взаимоотношения с некоторыми объектами. В данном примере у процесса *A* есть доступ к потокам 1 и 2, а также доступ к мьютексам 1 и 2. У процесса *B* есть доступ к потоку 3 и к мьютексам 2 и 3. В соответствующих элементах таблицы дескрипторов хранятся права доступа к каждому из этих объектов. Так, у процесса *A* есть права блокировать и разблокировать его мьютексы, но нет права уничтожить их. Обратите внимание, что мьютекс 2 используется совместно двумя процессами, обеспечивая синхронизацию их потоков. Остальные мьютексы не являются совместно используемыми. Это может означать, что потоки процесса *A* используют для своей внутренней синхронизации мьютекс 1, а потоки процесса *B* для своей внутренней синхронизации пользуются мьютексом 3.

Пространство имен объектов

По мере того как во время выполнения программы создаются и удаляются объекты, менеджеру объектов необходимо следить за ними. Для выполнения этой работы он поддерживает пространство имен, в котором располагаются все объекты системы. Пространство имен может использоваться процессом, чтобы найти и открыть дескриптор объекта другого процесса при условии, что для этого у него есть необходимые разрешения. Пространство имен объектов является одним из трех пространств имен, поддерживаемых в Windows 2000. Остальные два представляют собой пространство имен файловой системы и пространство имен реестра. Все три являются иерархическими пространствами имен со множеством уровней каталогов для организации элементов. Объекты каталогов, упомянутые в табл. 11.6, предоставляют средства реализации этого иерархического пространства имен для объектов.

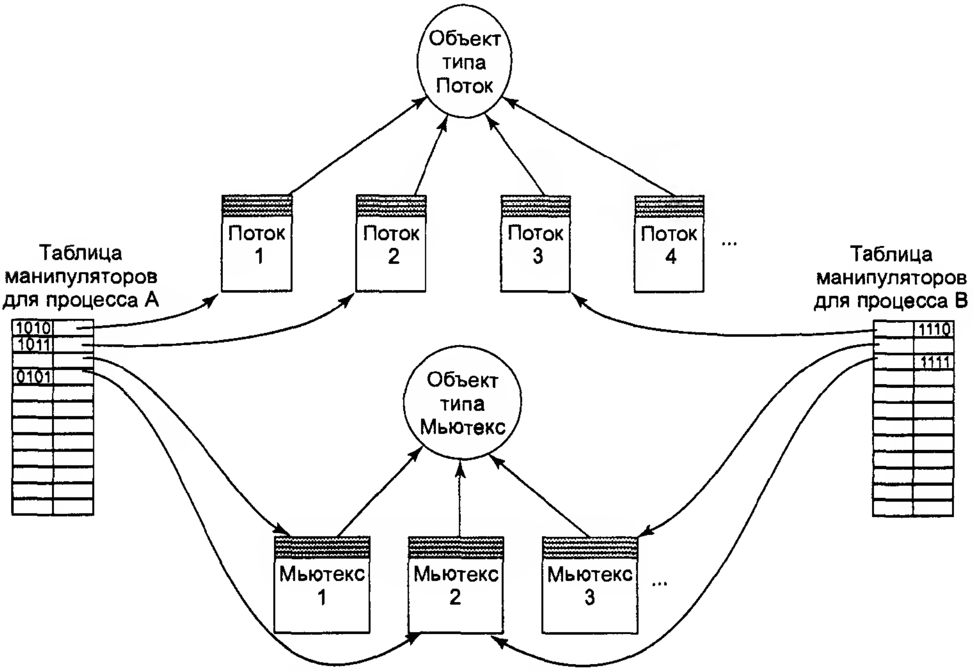


Рис. 11.5. Взаимоотношения между таблицами дескрипторов, объектами и типовыми объектами

Поскольку объекты исполняющей системы являются временными (то есть исчезают при выключении компьютера, в отличие от файловой системы и элементов реестра), в начале загрузки системы в памяти нет объектов и пространство имен объектов пусто. Во время загрузки различные части исполняющей системы создают каталоги и заполняют их объектами. Например, когда менеджер plug-and-play обнаруживает новые устройства, он создает по объекту для каждого устройства и помещает эти объекты в пространство имен. Когда система полностью загружена, все устройства ввода-вывода, дисковые разделы и другие интересные открытия системы оказываются в пространстве имен объектов.

Не все объекты создаются по методу Колумба — просто отправиться посмотреть, что удастся найти. Некоторые компоненты исполняющей системы смотрят в реестр, чтобы определить, что им делать. Важнейший пример — драйверы устройств. При загрузке система смотрит в реестр, чтобы узнать, какие драйверы ей нужны. При загрузке каждого драйвера создается объект, а его имя добавляется в пространство имен объектов. В системе обращение к драйверу осуществляется по указателю на его объект.

Хотя пространство имен объектов является центральным для всего функционирования системы, почти никому не известно о его существовании, потому что без специальных средств просмотра оно невидимо для пользователей. Одним из таких инструментов просмотра является программа *winobj*, которую можно бесплатно получить на сайте www.sysinternals.com. При запуске эта программа отображает пространство имен объектов, как правило, содержащее каталоги объектов, некоторые из них перечислены в табл. 11.7.

Таблица 11.7. Некоторые типичные каталоги пространства имен объектов

Каталог	Содержание
??	Начальное место для поиска устройств MS-DOS, например, C:
Device	Все обнаруженные устройства ввода-вывода
Driver	Объекты, соответствующие каждому загруженному драйверу устройства
ObjectTypes	Объекты типов, показанные на рис. 11.5
Windows	Объекты для отправки сообщений всем окнам
BaseNamedObjs	Объекты, создаваемые пользователем, такие как семафоры, мьютексы и т. д.
Arcname	Имена разделов, обнаруженные загрузчиком
NLS	Объекты языковой поддержки
FileSystem	Объекты драйверов файловой системы и объекты распознавателя файловой системы
Security	Объекты системы безопасности
KnownDLLs	Совместно используемые библиотеки, находящиеся в открытом состоянии

Может показаться несколько странным, что каталог, названный *??*, содержит все имена устройств стиля MS-DOS, такие как *A*: для гибкого диска и *C*: для первого жесткого диска. Эти имена в действительности представляют собой символьные ссылки на каталог *Device*, в котором располагаются объекты устройств. Имя *??* было выбрано, чтобы оно оказывалось первым в алфавитном порядке для ускорения поиска всех путей, начинающихся с буквы привода. Содержимое остальных каталогов должно говорить само за себя.

Подсистемы окружения

Возвращаясь к рис. 11.2, мы видим, что операционная система Windows 2000 состоит из компонентов, работающих в режиме ядра, и компонентов, работающих в режиме пользователя. Мы закончили наше изучение компонентов, работающих в режиме ядра, теперь пора перейти к обсуждению компонентов, работающих в режиме пользователя. Существует три типа таких компонентов: динамические библиотеки DLL, подсистемы окружения и служебные процессы. Эти компоненты работают вместе, предоставляя каждому пользовательскому процессу интерфейс, отличный от интерфейса системных вызовов Windows 2000.

Операционной системой Windows 2000 поддерживаются три различных документированных интерфейса прикладного программирования API: Win32, POSIX и OS/2. У каждого из этих интерфейсов есть список библиотечных вызовов, которые могут использовать программисты. Работа библиотек DLL (Dynamic Link Library — динамически подключаемая библиотека) и подсистем окружения заключается в том, чтобы реализовать функциональные возможности опубликованного интерфейса, тем самым скрывая истинный интерфейс системных вызовов от прикладных программ. В частности, интерфейс Win32 является официальным интерфейсом для операционных систем Windows 2000, Windows NT, Windows 95/98/Me и, в некоторой степени, для Windows CE. При использовании библиотеки DLL и подсистемы окружения Win32 программа может быть написана в соответствии со спецификацией Win32, в результате чего она сможет без каких-либо

изменений работать на всех этих версиях Windows, несмотря на то, что сами системные вызовы в различных системах различны.

Рассмотрим способ реализации этих интерфейсов на примере Win32. Программа, пользующаяся интерфейсом Win32, как правило, состоит из большого количества обращений к функциям Win32 API, например `CreateWindow`, `DrawMenuBar` и `OpenSemaphore`. Существуют тысячи подобных вызовов, и большинство программ использует значительное их количество. Один из возможных способов реализации заключается в статическом связывании каждой программы, использующей интерфейс Win32, со всеми библиотечными процедурами, которыми она пользуется. При таком подходе каждая двоичная программа будет содержать копию всех используемых ею процедур в своем исполняемом двоичном файле.

Недостаток такого подхода заключается в том, что при этом расходуется много памяти, если пользователь одновременно откроет несколько программ, использующих одни и те же библиотечные процедуры. Например, программы *Word*, *Excel* и *Powerpoint* используют абсолютно одинаковые процедуры для открытия диалоговых окон, рисования окон, отображения меню, работы с буфером обмена и т. д. Поэтому, если одновременно открыть все эти программы, при такой реализации программ в памяти будут находиться три (идентичные) копии каждой библиотечной процедуры.

Чтобы избежать подобной проблемы, все версии Windows поддерживают динамические библиотеки, называемые **DLL** (Dynamic-Link Library — динамически подсоединяемая библиотека). Каждая динамическая библиотека содержит набор тесно связанных библиотечных процедур и все их структуры данных в одном файле, как правило (но не всегда), с расширением *.dll*. Когда приложение компонуется, компоновщик видит, что некоторые библиотечные процедуры принадлежат к динамическим библиотекам, и записывает эту информацию в заголовок исполняемого файла. Обращения к процедурам динамических библиотек производятся не напрямую, а при помощи вектора передачи в адресном пространстве вызывающего процесса. Изначально этот вектор заполнен нулями, так как адреса вызываемых процедур еще неизвестны.

При запуске прикладного процесса все требуемые динамические библиотеки обнаруживаются (на диске или в памяти) и отображаются на виртуальное адресное пространство процесса. Затем вектор передачи заполняется верными адресами, что позволяет вызывать библиотечные процедуры через этот вектор с незначительной потерей производительности. Выигрыш такой схемы заключается в том, что при запуске нескольких приложений, использующих одну и ту же динамическую библиотеку, в физической памяти требуется только одна копия текста DLL (но каждый процесс получает свою собственную копию приватных статических данных в DLL). В операционной системе Windows 2000 динамические библиотеки используются очень активно для всех аспектов системы.

Теперь мы достаточно подготовились к тому, чтобы понять, как реализован интерфейс Win32, а также другие интерфейсы. Каждый пользовательский процесс, как правило, связан с несколькими динамическими библиотеками, совместно реализующими интерфейс Win32. Чтобы обратиться к вызову API, вызывается одна из процедур в DLL (шаг *1* на рис. 11.6). Дальнейшие действия зависят от вызова Win32 API. Различные вызовы реализованы по-разному.

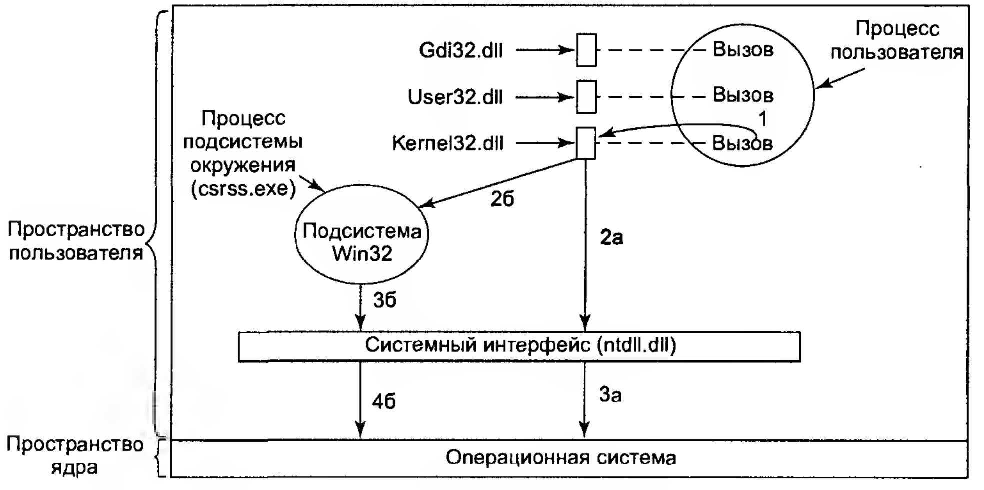


Рис. 11.6. Различные маршруты выполнения вызовов Win32 API

В некоторых случаях динамические библиотеки обращаются к другой динамической библиотеке (*ntdll.dll*) которая, в свою очередь, обращается к ядру операционной системы. Этот путь показан на рисунке как шаги *2а* и *3а*. Динамическая библиотека может также выполнить всю работу самостоятельно, совсем не обращаясь к системным вызовам. Для других вызовов Win32 API выбирается другой маршрут, а именно: сначала процессу подсистемы Win32 (*csrss.exe*) посылается сообщение, выполняющее некоторую работу, и обращающееся к системному вызову (шаги *2б*, *3б* и *4б*). При этом в некоторых случаях подсистема также выполняет всю работу в пространстве пользователя и немедленно возвращает управление. Передача сообщения между прикладным процессом и процессом подсистемы Win32 была старательно оптимизирована по времени, для чего был использован специальный механизм вызова локальной процедуры, реализованный в исполняющей системе и показанный как LPC на рис. 11.2.

В первой версии Windows NT практически все вызовы Win32 API выбирали маршрут *2б*, *3б*, *4б*, так как большая часть операционной системы (например, графика) была помещена в пространство пользователя. Однако, начиная с версии NT 4.0, для увеличения производительности большая часть кода была перенесена в ядро (в драйвер Win32/GDI на рис. 11.2). В Windows 2000 только небольшое количество вызовов Win32 API, например вызовы для создания процесса или потока, идут по длинному пути. Остальные вызовы выполняются напрямую, минуя подсистему окружения Win32.

Следует также сказать, что на рис. 11.6 показаны три наиболее важные DLL, но они не являются единственными динамическими библиотеками в системе. В каталоге *\winnt\system32* есть более 800 отдельных файлов DLL общим объемом в 130 Мбайт. Количество содержащихся в них вызовов API превышает 13 000. (В конце концов, 29 млн строк исходного текста должны были где-то храниться после компиляции.) Некоторые наиболее важные файлы динамических библиотек перечислены в табл. 11.8. Для каждого файла приведено количество экспортируемых функций (то есть видимых за пределами файла), но этот параметр меня-

ется со временем (увеличивается). Число экспортируемых функций в первом открытом выпуске *ntdll.dll* для Windows 2000 было равно 1179. Они представляют собой настоящие системные вызовы. 1209 вызовов, экспортируемых из файла *ntoskrnl.exe*, являются функциями, доступными для драйверов устройств и других программ, связанных с ядром. Вызовы в файле *win32k.sys* формально не экспортируются, так как файл *win32k.sys* не вызывается напрямую. Список экспортируемых функций в любом файле *.exe* или *.dll* можно просмотреть программой *depends*, входящей в пакет Platform SDK (Software Development Kit).

Таблица 11.8. Некоторые ключевые файлы Windows 2000, их режим работы, количество экспортируемых функций и основное содержание каждого файла

Файл	Режим	Количество функций	Содержание
hal.dll	Ядра	95	Низкоуровневое управление аппаратурой, например портами ввода-вывода
ntoskrnl.exe	Ядра	1209	Операционная система Windows 2000 (ядро + исполняющая система)
win32k.sys	Ядра	-	Множество системных вызовов, включая большую часть графики
ntdll.dll	Пользователя	1179	Диспетчер перехода из режима пользователя в режим ядра
csrss.exe	Пользователя	0	Процесс подсистемы окружения Win32
kernel32.dll	Пользователя	823	Большая часть системных вызовов ядра (неграфических)
gdi32.dll	Пользователя	543	Шрифт, текст, цвет, кисть, перо, растровые изображения, палитра, рисование и т. д.
user32.dll	Пользователя	695	Окна, значки, меню, курсоры, диалоговые окна, буфер обмена и т. д.
advapi32.dll	Пользователя	557	Защита, шифрование, реестр

Хотя интерфейс процессов Win32 является наиболее важным, в операционной системе Windows 2000 существует еще два интерфейса: POSIX и OS/2. Среда POSIX предоставляет минимальную поддержку для приложений UNIX. Она поддерживает практически только функции, описанные в стандарте P1003.1. Этим интерфейсом, например, не поддерживаются потоки, работа с окнами или сетью. Перенос любой реальной программы из системы UNIX в Windows 2000 при помощи этой подсистемы практически невозможен. Этот интерфейс был включен только потому, что некоторые министерства правительства США требовали, чтобы операционные системы для правительственных компьютеров были совместимы со стандартом P1003.1. Эта подсистема не является самодостаточной и пользуется вызовами подсистемы Win32 для большей части своей работы, но не предоставляя пользовательским программам полного интерфейса Win32 (если бы это было сделано корпорацией Microsoft, то подсистемой можно было бы пользоваться, причем для этого не потребовалось бы никаких специальных усилий).

Чтобы облегчить пользователям UNIX переход на Windows 2000, корпорация Microsoft также разработала программный продукт Interix, предоставляющий более высокую степень совместимости с системой UNIX, чем подсистема POSIX.

Функциональность подсистемы OS/2 ограничена практически в той же степени, что и функциональность подсистемы POSIX. Подсистема OS/2 также не поддерживает графические приложения. На практике она тоже полностью бесполезна. Таким образом, оригинальная идея наличия интерфейсов нескольких операционных систем, реализованных различными процессами в пространстве пользователя, окончилась ничем. Осталась лишь полная реализация интерфейса Win32 в режиме ядра.

Процессы и потоки в Windows 2000

В операционной системе Windows 2000 есть множество концепций для управления центральным процессором и объединения ресурсов. В следующих разделах мы их обсудим, рассмотрим некоторые соответствующие вызовы Win32 API, а также покажем, как эти концепции реализованы.

Основные понятия

В операционной системе Windows 2000 поддерживаются традиционные процессы, способные общаться и синхронизироваться друг с другом так же, как это делают процессы в UNIX. Каждый процесс содержит по крайней мере один поток, содержащий, в свою очередь, как минимум одно волокно (облегченный поток). Более того, для управления определенными ресурсами процессы могут объединяться в задания. Все вместе — задания, процессы, потоки и волокна — образует общий набор инструментов для управления ресурсами и реализации параллелизма как на однопроцессорных, так и на многопроцессорных машинах. Краткое описание этих четырех понятий приведено в табл. 11.9.

Таблица 11.9. Основные понятия, используемые для управления центральным процессором и ресурсами

Назввние	Описание
Задание	Набор процессов с общими квотами и лимитами
Процесс	Контейнер для ресурсов
Поток	Сущность, планируемая ядром
Волокно	Облегченный поток, управляемый полностью в пространстве пользователя

Рассмотрим по очереди эти понятия, начиная с самого крупного и заканчивая самым маленьким. **Задание** в Windows 2000 представляет собой набор, состоящий из одного или нескольких процессов, управляемых как единое целое. В частности, с каждым заданием ассоциированы квоты и лимиты ресурсов, хранящиеся в соответствующем объекте задания. Квоты включают такие пункты, как максимальное количество процессов (не позволяющее процессам задания создавать бесконтрольное количество дочерних процессов), суммарное время центрального процессора, доступное для каждого процесса в отдельности и для всех процессов вместе, а также максимальное количество используемой памяти для процесса и для всего задания.

Задания также могут ограничивать свои процессы в вопросах безопасности, например, запрещать им получать права администратора (суперпользователя) даже при наличии правильного пароля.

Процессы являются более интересными, чем задания, а также более важными. Как и в системе UNIX, процессы представляют собой контейнеры для ресурсов. У каждого процесса есть 4-гигабайтное адресное пространство, в котором пользователь занимает нижние 2 Гбайт (в версиях Windows 2000 Advanced Server и Datacenter Server этот размер может быть по желанию увеличен до 3 Гбайт), а операционная система занимает остальную его часть. Таким образом, операционная система присутствует в адресном пространстве каждого процесса, хотя она и защищена от изменений с помощью аппаратного блока управления памятью MMU. У процесса есть идентификатор процесса, один или несколько потоков, список дескрипторов (управляемых в режиме ядра) и маркер доступа, хранящий информацию защиты. Процессы создаются с помощью вызова Win32, который принимает на входе имя исполняемого файла, определяющего начальное содержимое адресного пространства, и создает первый поток.

Каждый процесс начинается с одного потока, но новые потоки могут создаваться динамически. Потоки формируют основу планирования центрального процессора, так как операционная система всегда для запуска выбирает поток, а не процесс. Соответственно, у каждого потока есть состояние (готовый, работающий, блокированный и т. д.), тогда как у процессов состояний нет. Потоки могут динамически создаваться вызовом Win32, которому в адресном пространстве процесса задается адрес начала исполнения. У каждого потока есть идентификатор потока, выбираемый из того же пространства, что и идентификаторы процессов, поэтому один и тот же идентификатор никогда не будет использован одновременно для процесса и для потока. Идентификаторы процессов и потоков кратны четырем, поэтому они могут использоваться в роли байтовых индексов в таблицах ядра, как и другие объекты.

Как правило, поток работает в пользовательском режиме, но когда он обращается к системному вызову, то переключается в режим ядра, после чего продолжает выполнять тот же поток, с теми же свойствами и ограничениями, которые были у него в режиме пользователя. У каждого потока есть два стека, один используется в режиме ядра, а другой в режиме пользователя. Помимо состояния, идентификатора и двух стеков, у каждого потока есть контекст (в котором сохраняются его регистры, когда он не работает), приватная область для локальных переменных, а также может быть свой собственный маркер доступа. Если у потока есть свой маркер доступа, то он перекрывает маркер доступа процесса, чтобы клиентские потоки могли передать свои права доступа серверным потокам, выполняющим работу для них. Когда поток завершает свою работу, он может прекратить свое существование. Когда прекращает существование последний активный поток, процесс завершается.

Важно понимать, что потоки представляют собой концепцию планирования, а не концепцию владения ресурсами. Любой поток может получить доступ ко всем объектам его процесса. Все, что ему для этого нужно сделать, — это заполучить дескриптор и обратиться к соответствующему вызову Win32. Для потока нет никаких ограничений доступа к объекту, связанных с тем, что этот объект создан или открыт другим потоком. Система даже не следит за тем, какой объект каким пото-

ком создан. Как только дескриптор объекта помещен в таблицу дескрипторов процесса, любой поток процесса может его использовать.

Помимо нормальных потоков, работающих в процессах пользователя, в операционной системе Windows 2000 есть множество процессов-демонов, не связанных ни с каким пользовательским процессом (они ассоциированы со специальной системой или простаивающими процессами). Некоторые демоны выполняют административные задачи, как, например, запись «грязных» страниц на диск, тогда как другие формируют пул, и ими могут пользоваться компоненты исполняющей системы или драйверы, которым нужно выполнить какие-либо асинхронные задачи в фоновом режиме. Некоторые из этих потоков будут рассмотрены позднее, когда мы дойдем до темы управления памятью.

Переключение потоков в операционной системе Windows 2000 занимает довольно много времени, так как для этого необходимо переключение в режим ядра, а затем возврат в режим пользователя. Для предоставления сильно облегченного псевдопараллелизма в Windows 2000 используются **волокна**, подобные потокам, но планируемые в пространстве пользователя создавшей их программой (или ее системой поддержки исполнения). У каждого потока может быть несколько волокон, так же как у процесса может быть несколько потоков, с той разницей, что когда волокно логически блокируется, оно помещается в очередь блокированных волокон, после чего для работы выбирается другое волокно в контексте того же потока.

Операционная система не знает о смене волокон, так как все тот же поток продолжает работу. Так как операционная система ничего не знает о волокнах, то с ними, в отличие от заданий, процессов и потоков, не связаны объекты исполняющей системы. Для управления волокнами нет и настоящих системных вызовов. Однако для этого есть вызовы Win32 API. Они относятся к тем вызовам Win32 API, которые не обращаются к системным вызовам, о чем уже рассказывалось при обсуждении рис. 11.6. Взаимосвязь между заданиями, процессами и потоками показана на рис. 11.7. (Несколько волокон могут объединяться в одном потоке, что не показано на рисунке.)

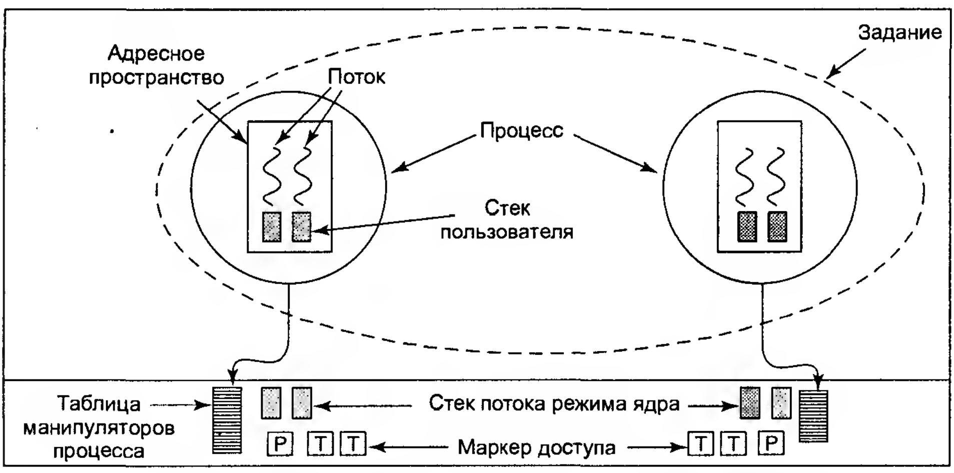


Рис. 11.7. Взаимосвязь между заданиями, процессами и потоками

Хотя мы не будем подробно обсуждать эту тему, следует скаэать, что операционная система Windows 2000 может работать на симметричных многопроцессорных системах. Это означает, что код операционной системы должен быть полностью реентерабельным, то есть каждая процедура должна быть написана таким образом, чтобы два или более центральных процессора могли поменять свои переменные без особых проблем. Во многих случаях это означает, что программные секции должны быть защищены при помощи спин-блокировки или мьютексов, удерживающих дополнительные центральные процессоры в режиме ожидания, пока первый центральный процессор не выполнит свою работу (при помощи последовательного доступа к критическим областям). Количество поддерживаемых системой процессоров управляется лицензионными ограничениями. Эти значения приведены в табл. 11.2. Нет никаких технических причин, по которым система Windows Professional не может работать на мультипроцессоре с 32 узлами — в конце концов, у нее тот же самый двоичный код, что и у версии Datacenter Server.

Верхний предел в 32 центральных процессора является жестким пределом, так как во многих местах операционной системы для учета использования центральных процессоров используются битовые массивы размером в 32-разрядное машинное слово. Например, один однословный битовый массив используется для того, чтобы следить, какой из центральных процессоров свободен в данный момент, а другой массив используется в каждом процессе для перечисления центральных процессоров, на которых этому процессу разрешено работать. 64-разрядная версия Windows 2000 должна будет без особых усилий поддерживать до 64 центральных процессоров. Для превышения этого ограничения потребуется существенная переделка программы (с использованием по нескольку слов для битовых массивов).

Вызовы API для управления заданиями, процессами, потоками и волокнами

Новые процессы создаются при помощи функции Win32 API `CreateProcess`. У этой функции 10 параметров, каждый из которых может задаваться в различных вариантах. Такая схема, очевидно, значительно сложнее схемы, применяемой в UNIX, в которой системный вызов `fork` вообще не имеет параметров, а у системного вызова `exec` их всего три: указатели на имя исполняемого файла, массив параметров командной (проанализированной) строки и строки окружения. Если не вдаваться в подробности, то у вызова `CreateProcess` следующие 10 параметров:

1. Указатель на имя исполняемого файла.
2. Сама командная строка (непроанализированная).
3. Указатель на описатель защиты процесса.
4. Указатель на описатель защиты для начального потока.
5. Бит, управляющий наследованием дескрипторов.
6. Разнообразные флаги (например, режим ошибки, приоритет, отладка, консоли).
7. Указатель на строки окружения.

8. Указатель на имя текущего рабочего каталога нового процесса.
9. Указатель на структуру, описывающую начальное окно на экране.
10. Указатель на структуру, возвращающую вызывающему процессу 18 значений.

В операционной системе Windows 2000 не поддерживается какой-либо иерархии процессов, например «родительский-дочерний». Все созданные процессы равны (не существует процессов, более равных, чем другие). Однако, поскольку один из 18 параметров, возвращаемых вызывающему процессу, представляет собой дескриптор нового процесса (что предоставляет контроль над новым процессом), существует негласная иерархия, заключающаяся в том, кто чьим дескриптором владеет. Хотя эти дескрипторы не могут напрямую передаваться другим процессам, у процесса есть способ создать дубликат дескриптора. Дубликат дескриптора может быть передан другому процессу и использоваться им, поэтому неявная иерархия процессов может просуществовать недолго.

Каждый процесс в Windows 2000 создается с одним потоком, но процесс может позднее создать дополнительные потоки. Создание потока проще создания процесса — у вызова `CreateThread` всего шесть параметров вместо десяти:

1. Описатель защиты (необязательный).
2. Начальный размер стека.
3. Адрес запуска.
4. Параметр, задаваемый пользователем.
5. Начальное состояние потока (готовый или блокированный).
6. Идентификатор потока.

Созданием потоков занимается ядро, поэтому оно имеет полное представление о потоках (потоки не реализуются полностью в пространстве пользователя, как это делается в некоторых других системах).

Межпроцессное взаимодействие

Для общения друг с другом потоки могут использовать широкий спектр возможностей, включая каналы, именованные каналы, почтовые ящики, вызов удаленной процедуры и совместно используемые файлы. Каналы могут работать в одном из двух режимов, выбираемом при создании канала: байтовом и режиме сообщений. Байтовые каналы работают так же, как и в системе UNIX. Каналы сообщений в чем-то похожи на байтовые каналы, но сохраняют границы между сообщениями, так что четыре записи по 128 байт будут читаться с другой стороны канала как четыре сообщения по 128 байт, а не как одно 512-байтовое сообщение, как это может случиться с байтовыми каналами. Также имеются именованные каналы, для которых существуют те же два режима. Именованные каналы, в отличие от обычных каналов, могут использоваться по сети.

Почтовые ящики представляют собой особенность системы Windows 2000, которой нет в UNIX. В некоторых аспектах они подобны каналам, но не во всем. Во-первых, почтовые ящики являются однонаправленными, тогда как каналы могут работать в обоих направлениях. Они также могут использоваться по сети, но не предоставляют гарантированной доставки. Наконец, они позволяют

отправляющему процессу использовать широковещание для рассылки сообщения не одному, а сразу многим получателям.

Сокеты подобны каналам с тем отличием, что они при нормальном использовании соединяют процессы на разных машинах. Например, один процесс пишет в сокет, а другой процесс на удаленной машине читает из него. Сокеты также могут использоваться для соединения процессов на одной машине, но поскольку их использование влечет за собой большие накладные расходы, чем использование каналов, то, как правило, они применяются в контексте сети.

Вызов удаленной процедуры представляет собой тот способ, которым процесс *A* просит процесс *B* вызвать процедуру в адресном пространстве процесса *B* от имени процесса *A* и вернуть результат процессу *A*. Существуют различные ограничения на параметры. Например, нет смысла передавать указатель другому процессу.

Наконец, процессы могут совместно использовать память для одновременного отображения одного и того же файла. Все, что один процесс будет писать в этот файл, будет появляться в адресном пространстве других процессов. С помощью такого механизма можно легко реализовать общий буфер, применяемый в задаче производителя и потребителя.

Помимо многочисленных механизмов межпроцессного взаимодействия, операционная система Windows 2000 также предоставляет множество механизмов синхронизации, включая семафоры, мьютексы, критические регионы и события. Все эти механизмы работают с потоками, а не процессами, так что когда поток блокируется на семафоре, другие потоки этого процесса (если такие есть) не затрагиваются и могут продолжать работу.

Семафор создается при помощи API-функции `CreateSemaphore`, которая может задать для него начальное значение, а также установить максимальное значение. Семафоры представляют собой объекты в ядре и, таким образом, обладают дескрипторами или дескрипторами защиты. Копия дескриптора может быть получена с помощью функции `DuplicateHandle` и передана другому процессу, в результате чего несколько процессов могут синхронизироваться, используя один семафор. Существуют вызовы для выполнения на семафоре операций `up` и `down`, они имеют несколько странные имена: `ReleaseSemaphore` (`up`) и `WaitForSingleObject` (`down`). Функции `WaitForSingleObject` также можно задать интервал времени, по истечении которого ждущий изменения состояния семафора поток будет отпущен, даже если семафор останется равным 0 (хотя использование таймеров приводит к конфликтам).

Мьютексы также представляют собой объекты ядра, используемые для синхронизации, но они проще семафоров, так как не содержат счетчиков. По существу, они являются блокировками, для работы с которыми используются функции API `WaitForSingleObject` и `ReleaseMutex`. Как и дескрипторы семафоров, дескрипторы мьютексов можно скопировать и передать другому процессу, так что потоки различных процессов смогут иметь доступ к одному и тому же мьютексу.

Третий механизм синхронизации основан на **критических секциях** (которые назывались критическими областями в других главах этой книги). Критические секции подобны мьютексам, но отличаются тем, что они связаны с адресным пространством создавшего их потока. Поскольку критические секции не являются объектами ядра, у них нет дескрипторов или дескрипторов защиты и они не могут передаваться от одного процесса другому. Блокирование и разблокирование

выполняется функциями EnterCriticalSection и LeaveCriticalSection соответственно. Поскольку эти функции API в основном выполняются в пространстве пользователя и обращаются к системным вызовам в ядро, только когда требуется блокирование потока, они работают быстрее, чем мьютексы.

В последнем механизме синхронизации используются объекты ядра, называемые **событиями**, которые бывают двух видов: сбрасываемые вручную и сбрасываемые автоматически. Каждое событие может находиться в одном из двух состояний: установленном и сброшенном. Поток может ждать какого-либо события с помощью функции WaitForSingleObject. Если другой поток вызывает событие при помощи функции SetEvent, результат зависит от типа события. Если событие является сбрасываемым вручную, то все ждущие его потоки отпускаются, а событие остается в установленном состоянии, пока его кто-либо не сбросит при помощи функции ResetEvent. В случае сбрасываемого автоматически события отпускается только один ожидающий его поток, а событие тут же сбрасывается. Кроме функции SetEvent существует также функция PulseEvent, отличающаяся от первой функции тем, что если этого события никто не ждет, событие все равно само сбрасывается и, таким образом, пропадает впустую. При использовании функции SetEvent событие, которого никто не ждет, напротив, остается в установленном состоянии, так что как только какой-либо поток обратится к функции WaitForSingleObject, он будет тут же отпущен, после чего событие сбросится.

События, мьютексы и семафоры могут иметь имена и храниться в файловой системе, подобно именованным каналам. Несколько процессов могут синхронизироваться друг с другом, открывая одно и то же событие, мьютекс или семафор, что проще, чем создание такого объекта одним процессом и передача другим процессам дубликата дескриптора, хотя такой способ, конечно, также возможен.

Интерфейс Win32 API содержит около 100 вызовов, работающих с процессами, потоками и волокнами. Значительное количество этих вызовов в той или иной мере имеет отношение к межпроцессному взаимодействию. Некоторые из обсуждавшихся выше функций, а также некоторые другие важные функции приведены в табл. 11.10.

Таблица 11.10. Некоторые функции Win32 API для управления процессами, потоками и волокнами

Функция Win32 API	Описание
CreateProcess	Создать новый процесс
CreateThread	Создать новый поток в существующем процессе
CreateFiber	Создать новое волокно
ExitProcess	Завершить текущий процесс и все его потоки
ExitThread	Завершить этот поток
ExitFiber	Завершить это волокно
SetPriorityClass	Задать класс приоритета для процесса
SetThreadPriority	Задать приоритет для потока
CreateSemaphore	Создать новый семафор
CreateMutex	Создать новый мьютекс
OpenSemaphore	Открыть существующий семафор

продолжение ⇒

Таблица 11.10 (*продолжение*)

Функция Win32 API	Описание
OpenMutex	Открыть существующий мьютекс
WaitForSingleObject	Блокироваться на одном семафоре, мьютексе и т. д.
WaitForMultipleObjects	Блокироваться на множестве объектов, чьи дескрипторы перечисляются
PulseEvent	Перевести событие в сигнализирующее состояние, а затем вернуть в несигнализирующее
ReleaseMutex	Освободить мьютекс, чтобы другой поток мог его захватить
ReleaseSemaphore	Увеличить на единицу счетчик семафора
EnterCriticalSection	Захватить блокировку для критической секции
LeaveCriticalSection	Освободить блокировку для критической секции

Большинство вызовов из табл. 11.10 либо обсуждались выше, либо должны говорить сами за себя. Снова обращаю ваше внимание, что не все они являются системными вызовами. Как уже упоминалось, операционная система Windows 2000 ничего не знает о волокнах. Они полностью реализованы в пространстве пользователя. Поэтому функция `CreateFiber` выполняет свою работу целиком в пространстве пользователя, не обращаясь к системным вызовам (разве что только с просьбой выделить ей память). Многие другие вызовы Win32 также обладают подобным свойством, включая уже упомянутые функции `EnterCriticalSection` и `LeaveCriticalSection`.

Реализация процессов и потоков

Процессы и потоки имеют большее значение и являются более сложными, чем задания и волокна, поэтому мы сконцентрируем наше внимание на них. Процесс создается другим процессом при помощи вызова интерфейса Win32 `CreateProcess`. Этот вызов обращается (в режиме пользователя) к процедуре в динамической библиотеке *kernel32.dll*, которая в несколько этапов создает процесс, используя при этом множество системных вызовов и других действий.

1. Указанный в качестве параметра исполняемый файл изучается и открывается. Если это корректный исполняемый файл формата POSIX, OS/2, 16-разрядной системы Windows или MS-DOS, то для него устанавливается специальное окружение. Если это корректный *.exe* файл 32-разрядного интерфейса Win32, в реестре проверяется, не является ли этот файл особенным в каком-либо смысле (например, должен выполняться под контролем отладчика). Все эти действия выполняются в режиме пользователя внутри *kernel32.dll*.
2. Выполняется обращение к системному вызову `NtCreateProcess`, чтобы создать пустой объект процесса и поместить его в пространство менеджера объектов. Создаются объект ядра и объект исполняющей системы. Кроме того, менеджер процессов создает для объекта управляющий блок процесса и инициализирует его идентификатором процесса, квотами, маркером доступа и различными другими полями. Также создается объект секции, чтобы следить за адресным пространством процесса.
3. Когда динамическая библиотека *kernel32.dll* снова получает управление, она обращается к еще одному системному вызову, `NtCreateThread`, чтобы создать

начальный поток. Для потока создаются стек режима пользователя и стек режима ядра. Размер стека указан в заголовке исполняемого файла.

4. Затем библиотека *kernel32.dll* посылает подсистеме окружения Win32 сообщение, в котором содержится информация о новом процессе, и передает ей дескрипторы процесса и потока. Данные о процессе и потоке помещаются в таблицы подсистемы, в результате чего у нее оказывается полный список всех процессов и потоков. Затем подсистема отображает курсор в виде стрелки с песочными часами, чтобы сообщить пользователю, что что-то происходит, но курсор может использоваться. Когда процесс обращается к своему первому вызову графического интерфейса пользователя, как правило, чтобы создать окно, курсор удаляется (если обращения к вызову не поступает, курсор удаляется через 2 с).
5. В этот момент поток готов к работе. Он начинается с запуска процедуры системы поддержки исполнения программ для завершения инициализации.
6. Процедура системы поддержки исполнения программ устанавливает приоритет потока, сообщает загруженным библиотекам DLL о появлении нового потока, а также выполняет другую рутинную работу. Наконец, она запускает код основной программы потока.

Создание потока также состоит из нескольких этапов, но мы не будем вдаваться в эти подробности. Сначала работающий процесс обращается к функции `CreateThread`, которая вызывает процедуру внутри *kernel32.dll*. Эта процедура выделяет в вызывающем процессе память для стека режима пользователя, а затем обращается к системному вызову `NtCreateThread`, чтобы создать объект потока для исполняющей системы, проинициализировать его, а также создать и проинициализировать блок управления потоком. И в этом случае уведомляется подсистема Win32, а информация о потоке помещается в ее таблицы. Затем поток начинает работу с собственной инициализации.

Когда создается процесс или поток, исходному процессу возвращается дескриптор, который можно использовать для запуска, остановки, уничтожения и проверки созданного процесса или потока. Владелец дескриптора может передать его другому процессу защищенным способом. Эта техника применяется, чтобы отладчики могли иметь полный контроль над управляемыми ими процессами.

Планирование

В операционной системе Windows 2000 нет центрального потока планирования. Вместо этого, когда какой-либо поток не может более выполняться, этот поток сам переходит в режим ядра и запускает планировщика, чтобы определить, на какой поток переключиться. Текущий поток выполняет программу планировщика при одном из следующих условий:

1) поток блокируется на семафоре, мьютексе, событии, операции ввода-вывода и т. д;
2) поток сигнализирует каким-либо объектом (например, выполняет операцию `up` на семафоре);
3) истекает квант времени работающего потока.

В случае 1 поток уже работает в режиме ядра, чтобы выполнить операцию с объектом диспетчера или ввода-вывода. Возможно, он не может продолжать работу, поэтому он должен сохранить свой контекст, запустить программу планировщика, чтобы выбрать своего преемника, и загрузить контекст этого потока, чтобы запустить его.

В случае 2 поток также находится в ядре. Однако после сигнализирования объектом он, определенно, может продолжать работу, так как эта операция никогда не приводит к блокированию. Тем не менее поток должен запустить процедуру планировщика, чтобы посмотреть, нет ли среди готовых к работе потока с более высоким приоритетом. Если такой поток есть, происходит переключение на этот поток, так как операционная система Windows 2000 является системой с приоритетным прерыванием (то есть переключение потока может произойти в любой момент, а не только тогда, когда у текущего потока закончится выделенный ему квант времени).

В случае 3 происходит эмулированное прерывание с передачей управления в ядро. При этом поток также запускает процедуру планировщика, чтобы определить, какой поток следует запустить после текущего потока. Если все остальные потоки в данный момент окажутся заблокированными, планировщик может продолжить выполнение текущего потока, выделив ему новый квант времени. В противном случае происходит переключение потока.

Планировщик также вызывается при еще двух условиях:

1. Завершается операция ввода-вывода.
2. Истекает ожидание таймера.

В первом случае какой-нибудь поток, возможно, ожидал окончания этой операции ввода-вывода и теперь может продолжить свою работу. Необходимо определить, должен ли этот поток прервать выполнение текущего потока, так как потокам не гарантируется минимальный рабочий интервал времени. Планировщик не запускается во время работы самой процедуры обработки прерываний (так как при этом прерывания могут оказаться запрещенными на слишком долгий срок). Вместо этого отложенный вызов процедуры DPC устанавливается в очередь и выполняется немного позднее, после того как процедура обработки прерываний закончит свою работу. Во втором случае поток выполнил операцию `down` на семафоре или блокировался на каком-либо другом объекте, но установленное время ожидания истекло. И в этом случае обработчик прерываний должен установить DPC в очередь, чтобы она не была запущена во время работы обработчика прерываний. Если в результате тайм-аута поток оказался готовым к работе, будет запущен планировщик, и если ничего более важного в данный момент нет, будет выполнен отложенный вызов процедуры.

Теперь мы подходим к самому алгоритму планирования. Интерфейс Win32 API содержит два вызова, предоставляющих процессам возможность влиять на планирование потоками. Алгоритм планирования в большой степени определяется этими вызовами. Во-первых, есть вызов `SetPriorityClass`, устанавливающий класс приоритета всех потоков вызывающего процесса. К допустимым значениям приоритета относятся: реального времени, высокий, выше нормы, нормальный, ниже нормы и неработающий.

Во-вторых, имеется вызов SetThreadPriority, устанавливающий относительный приоритет некоторого потока (возможно, но не обязательно, потока, обращающегося к этому вызову) по сравнению с другими потоками данного процесса. Приоритет может иметь следующие значения: критичный ко времени, самый высокий, выше нормы, нормальный, ниже нормы, самый низкий и неработающий. Таким образом, шесть классов процессов и семь классов потоков могут образовать 42 комбинации. Эта информация поступает на вход алгоритма планирования.

Планировщик работает следующим образом. В системе существует 32 уровня приоритета, пронумерованные от 0 до 31. 42 комбинации отображаются на эти 32 приоритета в соответствии с табл. 11.11. Число в таблице определяет **базовый приоритет** потока. Кроме того, у каждого потока есть **текущий приоритет**, который может быть выше (но не ниже) базового приоритета и о котором мы скажем несколько слов.

Таблица 11.11. Преобразование приоритетов Win32 в приоритеты Windows 2000

Приоритеты потоков Win32	Классы приоритетов процессов Win32					
	Реального времени	Высокий	Выше нормы	Нормальный	Ниже нормы	Бездействующий
Критичный ко времени	31	15	15	15	15	15
Самый высокий	26	15	12	10	8	6
Выше нормы	25	14	11	9	7	5
Нормальный	24	13	10	8	6	4
Ниже нормы	23	12	9	7	5	3
Самый низкий	22	11	8	6	4	2
Неработающий	16	1	1	1	1	1

Чтобы использовать эти приоритеты для планирования, система содержит массив из 32 элементов, соответствующих приоритетам от 0 до 31, полученных из табл. 11.11. Каждый элемент массива указывает на начало списка готовых потоков с соответствующим приоритетом. Базовый алгоритм планирования состоит из процедуры сканирования массива от приоритета 31 до приоритета 0. Как только найден непустой элемент, выбирается поток в начале очереди и запускается на один квант времени. Когда квант истекает, поток направляется в конец очереди своего приоритета, а следующим выбирается поток в начале очереди. Другими словами, когда есть несколько готовых потоков с наивысшим уровнем приоритета, они запускаются поочередно, получая каждый по одному кванту времени. Если готовых потоков нет, запускается бездействующий поток.

Следует отметить, что при планировании не учитывается, какому процессу принадлежит тот или иной поток. То есть планировщик не выбирает сначала процесс, а затем поток в этом процессе. Он смотрит только на потоки. Он даже не знает, какой поток какому процессу принадлежит. На многопроцессорной системе каждый центральный процессор сам занимается планированием своих потоков при помощи массива приоритетов. Чтобы гарантировать, что в каждый момент времени лишь один центральный процессор работает с массивом, используется спин-блокировка.

Массив заголовков очередей показан на рис. 11.8. Из рисунка видно, что в действительности существует четыре категории приоритетов: реального времени, пользовательские, нулевой и бездействующий, равный –1. Об этом следует сказать несколько слов. Приоритеты с 16 по 31 называются приоритетами реального времени, но они таковыми не являются. Выполняющимся с этими приоритетами потокам не дается никаких гарантий и никакие сроки исполнения не учитываются. Это просто более высокие приоритеты, чем приоритеты с 0 по 15. Однако приоритеты с 16 по 31 зарезервированы для самой системы и для потоков, которым такой высокий приоритет явно задаст системный администратор. Обычные пользователи не могут запускать потоки со столь высокими приоритетами, и существует веская причина для этого. Если бы пользовательский процесс мог работать с приоритетом более высоким, чем, скажем, поток клавиатуры или мыши, то длительная работа такого высокоприоритетного потока без операций ввода-вывода (например, в цикле) повесила бы всю систему.

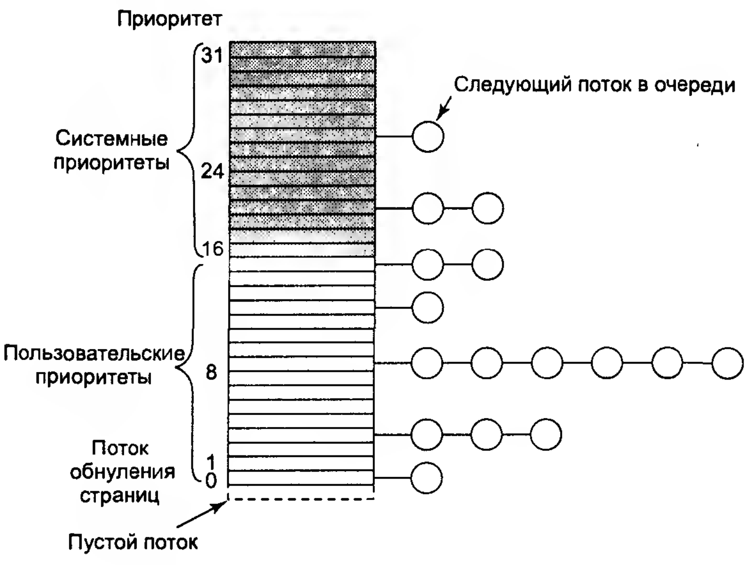


Рис. 11.8. Windows 2000 поддерживает 32 приоритета для потоков

Пользовательские потоки работают с приоритетами от 1 до 15. Устанавливая приоритеты процесса и потока, пользователь может отдавать преимущество тому или иному потоку. Нулевой поток работает в фоновом режиме и съедает все процессорное время, на которое больше никто не претендует. Его работа заключается в обнулении страниц для менеджера памяти. Его роль в системе будет обсуждаться ниже. Если и у этого потока нет работы, работает пустой поток. Однако он не является полноценным потоком.

Со временем для улучшения производительности системы в базовом алгоритме планирования было сделано несколько усовершенствований. При определенных условиях текущий приоритет пользовательского потока может быть поднят операционной системой выше базового приоритета, но никогда не может быть установлен выше приоритета 15. Так как массив на рис. 11.8 основан на текущем при-

оритете, изменение этого приоритета изменяет поведение алгоритма планирования. Для потоков с приоритетами 15 и выше никогда не делается никаких изменений приоритета.

Когда же увеличивается приоритет потока? Во-первых, когда завершается операция ввода-вывода и освобождает ожидающий ее поток, приоритет потока увеличивается, чтобы дать шанс этому потоку быстрее запуститься и снова запустить операцию ввода-вывода. Суть в том, чтобы поддерживать занятость устройств ввода-вывода. Величина, на которую увеличивается приоритет, зависит от устройства ввода-вывода. Как правило, это 1 для диска, 2 для последовательной линии, 6 для клавиатуры и 8 для звуковой карты.

Во-вторых, если поток ждал семафора, мьютекса или другого события, то когда он отпускается, к его приоритету прибавляется две единицы, если это поток переднего плана (то есть процесс, управляющий окном, которому в данный момент направляется ввод с клавиатуры), и одна единица в противном случае. Таким образом, интерактивный процесс получает преимущество перед большой толпой других процессов на уровне 8. Наконец, если поток графического интерфейса пользователя просыпается, потому что стал доступен оконный ввод, он также получает прибавку приоритета по той же самой причине.

Эти увеличения приоритета не вечны. Они незамедлительно вступают в силу, но если поток использует весь свой следующий квант времени, он теряет один пункт и перемещается на одну очередь вниз в массиве приоритетов. Если он использует еще один полный квант, он перемещается еще на один уровень ниже и т. д., пока он не достигнет своего базового уровня, где останется до тех пор, пока ему снова не будет прибавлен приоритет. Очевидно, если поток претендует на хорошее обслуживание, он должен действовать очень активно.

Есть еще один случай, при котором система изменяет приоритеты потоков. Представьте, что два потока работают вместе в задаче типа «производитель-потребитель». Работа производителя труднее, поэтому он получает более высокий приоритет, например 12, а потребитель получает приоритет 4. В определенный момент производитель заполняет до отказа общий буфер и блокируется на семафоре, как показано на рис. 11.9, *а*.

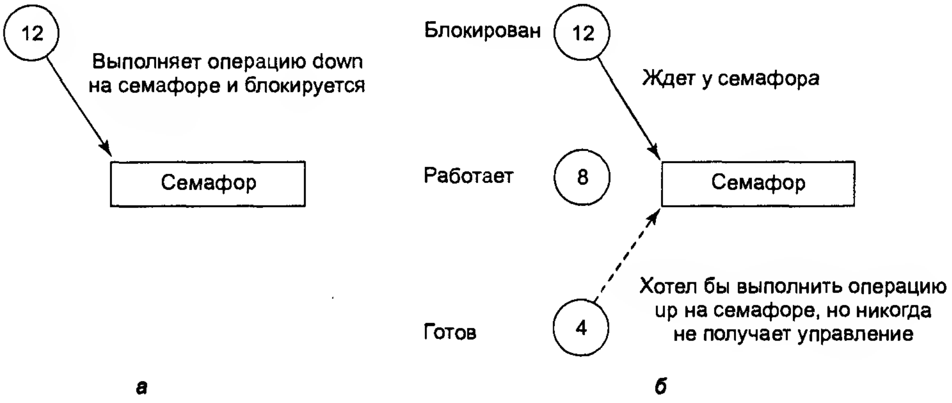


Рис. 11.9. Пример инверсии приоритета

Прежде чем потребитель снова получит шанс поработать, посторонний процесс с приоритетом 8 приходит в состояние готовности и получает управление, (рис. 11.9, *б*). Этот поток сможет работать столько, сколько захочет, так как его приоритет выше приоритета потребителя, а производитель, хоть и с большим приоритетом, заблокирован. При таких условиях производитель никогда снова не получит управления, пока поток с приоритетом 8 не остановится.

В операционной системе Windows 2000 эта проблема решается при помощи большой кувалды. Система следит, сколько времени прошло с тех пор, когда готовый к работе поток запускался в последний раз. Если какой-либо поток превысит определенный интервал времени, он получает на два кванта времени приоритет 15. Это может помочь разблокировать производителя. После двух квантов прибавка приоритета резко убирается. Вероятно, лучшим решением было бы наказывать потоки, которые полностью используют свои кванты снова и снова. В конце концов, проблему создает не поток, умирающий от голода, а жадный поток. Эта проблема хорошо известна под названием **инверсии приоритетов**.

Аналогичная проблема возникает в том случае, когда поток с приоритетом 16 захватывает мьютекс и долго не получает управления, в результате чего более важные системные потоки, ждущие этого мьютекса, умирают от истощения. Эту проблему можно устранить, если разрешить потоку, которому на короткое время требуется мьютекс, просто запрещать планирование на время использования мьютекса. На многопроцессорных системах следует применять спин-блокировку.

Напоследок следует сказать пару слов о кванте. В системе Windows 2000 Professional длительность кванта по умолчанию равна 20 мс; на однопроцессорных серверах его значение равно 120 мс; на многопроцессорных системах используются различные другие варианты в зависимости от частоты процессора. Более короткий квант улучшает работу интерактивных процессов, тогда как более длинный квант снижает количество переключений контекста и тем самым увеличивает производительность. Именно в этом смысл правой колонки табл. 11.2. Значения по умолчанию при желании могут быть увеличены в 2, 4 или 6 раз. Кстати, величина кванта была выбрана более десяти лет назад и не менялась с тех пор, хотя компьютеры за это время стали быстрее более чем на порядок. Вероятно, эти числа можно было бы без ущерба уменьшить в 5 или 10 раз и, возможно, получить при этом лучшее время отклика для интерактивных потоков в сильно загруженной системе.

Последняя модификация алгоритма планирования заключается в том, что когда окно становится окном переднего плана, все его потоки получают более длительные кванты времени. Величина прибавки интервала времени хранится в системном реестре. Таким образом, поток получает больше процессорного времени, и, соответственно, достигается лучшее обслуживание для окна, перемещенного на передний план.

Эмуляция MS-DOS

Одна из задач проектирования системы Windows 2000 была унаследована от NT: постараться поддержать по возможности большое (в разумных пределах) количество программ для MS-DOS. Эта цель принципиально отличалась от задачи,

поставленной перед создателями системы Windows 98: в ней должны были работать все старые программы для MS-DOS (от себя добавим: неважно, насколько плохо они работали).

Операционная система Windows 2000 запускает старые программы в полностью защищенном окружении. Когда запускается программа, написанная для системы MS-DOS, запускается нормальный процесс Win32, в который загружается эмулятор MS-DOS *ntvdm* (NT Virtual DOS Machine – виртуальная машина DOS для NT), сканирующий программу MS-DOS и выполняющий ее системные вызовы. В системе MS-DOS всего лишь 1 Мбайт оперативной памяти на процессоре Intel 8088 и до 16 Мбайт с переключением банков и другими трюками на процессоре Intel 80286. Поэтому можно без особого риска поместить процесс *ntvdm* в старшие адреса виртуального адресного пространства, где программа не сможет к нему адресоваться. Эта ситуация показана на рис. 11.10.

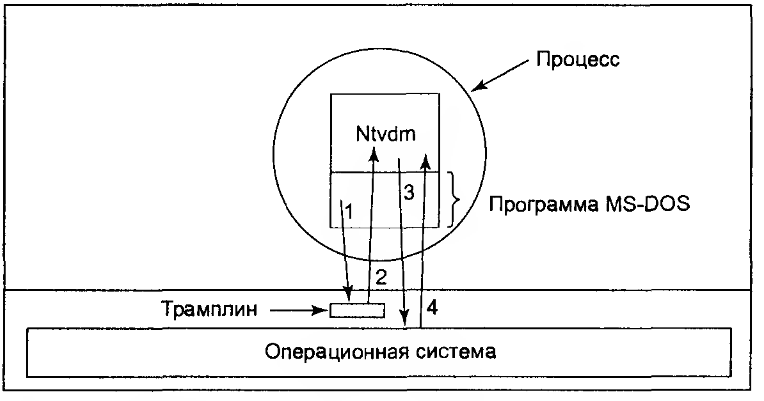


Рис. 11.10. Запуск старых программ для MS-DOS в системе Windows 2000

Когда программа MS-DOS выполняет обычные команды процессора, она может работать на голой аппаратуре, так как процессор Pentium полностью поддерживает наборы команд процессоров Intel 8088 и Intel 80286. Самое интересное начинается, когда программа MS-DOS собирается выполнить операцию ввода-вывода или обратиться к операционной системе. Корректно написанная программа просто выполняет системный вызов. Рассчитывая на корректное поведение, эмулятор *ntvdm* инструктирует систему Windows 2000 перенаправлять все системные вызовы ему. В результате системный вызов просто «отскакивает» от операционной системы и перехватывается эмулятором (шаги 1 и 2 на рис. 11.10). Такой метод иногда называют использованием **трамплина** (батута).

Получив управление, эмулятор определяет, что пытается сделать программа, и обращается к вызову Win32 для выполнения требуемой работы (шаги 3 и 4 на рис. 11.10). Пока программа ведет себя корректно и лишь обращается к легальным системным вызовам MS-DOS, этот метод прекрасно работает. Неприятность в том, что некоторые старые программы, написанные для работы в системе MS-DOS, обходят операционную систему и напрямую обращаются к видеопамяти, напрямую читают порты клавиатуры и т. д., то есть выполняют действия, недопустимые в защищенной среде. Если такое некорректное поведение программы приводит

к прерываниям, есть надежда, что эмулятор сумеет определить, чего хочет программа, и сможет эмулировать это действие. Если же определить намерения программы не удастся, программа просто уничтожается, так как задача стопроцентной эмуляции старых операционных систем никогда не ставилась при проектировании Windows 2000.

Загрузка Windows 2000

Прежде чем операционная система Windows 2000 сможет начать работу, она должна загрузиться. Процесс загрузки создает начальные процессы. В данном разделе мы кратко обсудим процесс загрузки операционной системы Windows 2000. С точки зрения аппаратного обеспечения, процесс загрузки состоит из чтения первого сектора первого диска (главной загрузочной записи), после чего прочитанной программе передается управление, как было описано в разделе «Форматирование дисков» главы 5. Эта короткая программа на ассемблере считывает таблицу разделов, чтобы определить, в каком разделе содержится загружаемая операционная система. Найдя раздел с операционной системой, начальный загрузчик считывает первый сектор этого раздела, называемый **загрузочным сектором**, и передает управление ему. Программа, содержащаяся в загрузочном секторе, считывает корневой каталог своего дискового раздела, находит в нем файл *ntldr* (еще одно археологическое свидетельство того, что Windows 2000 на самом деле представляет собой NT). Если этот файл удается найти, он загружается в память и ему передается управление. Программа *ntldr* загружает операционную систему Windows 2000. Существует несколько версий загрузочного сектора в зависимости от формата раздела (FAT-16, FAT-32 или NTFS). При установке Windows 2000 на диск записываются соответствующие версии главной загрузочной записи и загрузочного сектора.

Затем программа *ntldr* считывает файл *Boot.ini*, представляющий собой единственный файл с информацией о конфигурации, не содержащейся в реестре. Он хранит в себе списки всех версий файлов *hal.dll* и *ntoskernl.exe*, которые могут быть загружены с данного раздела диска. В этом файле также содержатся такие параметры, как количество центральных процессоров и оперативной памяти, сколько памяти отводить процессу пользователя (2 или 3 Гбайт), а также на какой частоте работают часы реального времени. Затем программа *ntldr* выбирает и загружает файлы *hal.dll* и *ntoskernl.exe*, а также файл *bootvid.dll*, представляющий собой видеодрайвер по умолчанию. Он обеспечивает вывод на дисплей во время процесса загрузки. После этого программа *ntldr* считывает реестр, чтобы найти драйверы, необходимые для завершения загрузки (например, драйверы клавиатуры и мыши, а также десятки других драйверов, требуемых для управления различными микросхемами на материнской плате). Наконец, загрузчик считывает все эти драйверы и передает управление программе *ntoskernl.exe*.

После запуска операционная система выполняет некоторые общие процедуры инициализации, а затем вызывает компоненты исполняющей системы, чтобы те также выполнили собственную инициализацию. Например, менеджер объектов подготавливает свое пространство имен, чтобы другие компоненты могли обращаться к нему и добавлять свои объекты в пространство имен. Многие компоненты также выполняют определенные действия, относящиеся к их функциям, так,

менеджер памяти настраивает начальные таблицы страниц, а менеджер plug-and-play определяет, какие устройства ввода-вывода присутствуют, и загружает их драйверы. Вся загрузка состоит из десятков этапов, в течение которых на экране отображается полоса прогресса, растущая по мере выполнения очередных этапов. Последний этап заключается в создании первого настоящего пользовательского процесса, **сеансового менеджера** *smss.exe*. Как только этот процесс начинает работу, загрузка считается законченной.

Сеансовый менеджер представляет собой «родной» процесс операционной системы Windows 2000. Он обращается к истинным системным вызовам и не пользуется вызовами подсистемы окружения Win32, которая в тот момент еще даже не работает. Одной из его первоочередных обязанностей является запуск этой подсистемы (*csrss.exe*). Он также считывает с диска ульи реестра и узнает из них, что еще он должен сделать. Как правило, его работа заключается в помещении множества объектов в пространство имен менеджера объектов, создании дополнительных файлов подкачки и открытии нужных DLL. Завершив свою работу, сеансовый менеджер создает демон регистрации *winlogon.exe*.

В этот момент операционная система загружена и работает. Теперь пора запустить служебные процессы (демоны в пространстве пользователя) и позволить пользователям регистрироваться в системе. Сначала *winlogon.exe* создает менеджера аутентификации (*lsass.exe*), а затем запускает родительский процесс всех служебных процессов (*services.exe*). Последний процесс по информации, хранящейся в реестре, определяет, какие демоны в пространстве пользователя нужно запустить и в каких файлах они находятся. После этого он приступает к их созданию. Такие демоны показаны на рис. 11.2. Как правило, уже после того, как первый пользователь зарегистрировался в системе, но еще до того, как он успел в ней что-либо сделать, в операционной системе Windows 2000 наблюдается высокая активность с большим количеством обращений к диску. Это программа *services.exe* создает системные службы. Кроме того, она загружает все оставшиеся, еще не загруженные, драйверы устройств. Иерархия начальных процессов и некоторых типичных служб показана в табл. 11.12. Процессы, расположенные выше линии, запускаются всегда. Процессы, помещенные в нижней части таблицы, представляют собой примеры служб, которые могут быть запущены.

Программа *winlogon.exe* также отвечает за регистрацию всех пользователей в системе. Диалоговое окно отображается отдельной программой *msgina.dll*, чтобы другие производители программного обеспечения могли заменить стандартную процедуру регистрации с вводом имени и пароля идентификацией отпечатков пальцев или еще чем-нибудь. После успешного входа пользователя в систему программа *winlogon.exe* получает из реестра профиль пользователя и определяет по нему, какую оболочку запустить. Многие пользователи этого не осознают, но стандартный рабочий стол Windows представляет собой просто программу *explorer.exe* (Проводник), у которой настроены некоторые параметры. При желании пользователь может выбрать в качестве оболочки другую программу, включая командную строку или даже редактор *Word*, для чего ему нужно просто отредактировать реестр. Однако редактирование реестра — занятие не для слабых духом. Одна ошибка может сделать систему незагружаемой.

Таблица 11.12. Процессы, запускаемые на стадии загрузки

Процесс	Описание
idle	Не настоящий процесс, а жилище пустого потока
system	Создает smss.exe и файлы подкачки, читает реестр, открывает DLL
smss.exe	Первый истинный процесс; много инициализации; создает csrss и winlogon
csrss.exe	Процесс подсистемы Win32
winlogon.exe	Демон регистрации
lsass.exe	Менеджер аутентификации
services.exe	Смотрит в реестр и запускает службы
Сервер принтера	Позволяет удаленному заданию использовать принтер
Файловый сервер	Обслуживает запросы к локальным файлам
Демон telnet	Обеспечивает удаленную регистрацию
Обработчик входящей электронной почты	Принимает и хранит входящую почту
Обработчик входящих факсов	Принимает и печатает входящие факсы
Распознаватель DNS	Сервер службы DNS Интернета
Регистратор событий	Регистрирует различные события системы
Менеджер plug-and-play	Сканирует аппаратуру, чтобы определить устройства, присутствующие в системе

Управление памятью

В операционной системе Windows 2000 крайне сложная система виртуальной памяти. В Windows 2000 есть множество функций Win32 для использования виртуальной памяти и часть исполняющей системы плюс шесть выделенных потоков ядра для управления ею. В следующих разделах мы рассмотрим основные понятия, вызовы Win32 и, наконец, реализацию управления памятью.

Основные понятия

В операционной системе Windows 2000 у каждого пользовательского процесса есть собственное виртуальное адресное пространство. Виртуальные адреса 32-разрядные, поэтому у каждого процесса 4 Гбайт виртуального адресного пространства. Нижние 2 Гбайт за вычетом около 256 Мбайт доступны для программы и данных процесса; верхние 2 Гбайт защищенным образом отображаются на память ядра. Страницы виртуального адресного пространства имеют фиксированный размер (4 Кбайт на компьютере с процессором Pentium) и подгружаются по требованию.

Конфигурация виртуального адресного пространства для трех пользовательских процессов в слегка упрощенном виде показана на рис. 11.11. Белым цветом на рисунке изображена область приватных данных процесса. Затененные области представляют собой память, совместно используемую всеми процессами. Нижние и верхние 64 Кбайт каждого виртуального адресного пространства в обычном состо-

янии не отображаются на физическую память. Это делается преднамеренно, чтобы облегчить перехват программных ошибок. Недействительные указатели часто имеют значение 0 или –1, и попытки их использования в системе Windows 2000 вызовут немедленное прерывание вместо чтения или, что еще хуже, записи слова по неверному адресу. Однако когда запускаются старые программы MS-DOS в режиме эмуляции, нижние 64 Кбайт могут отображаться на физическую память.

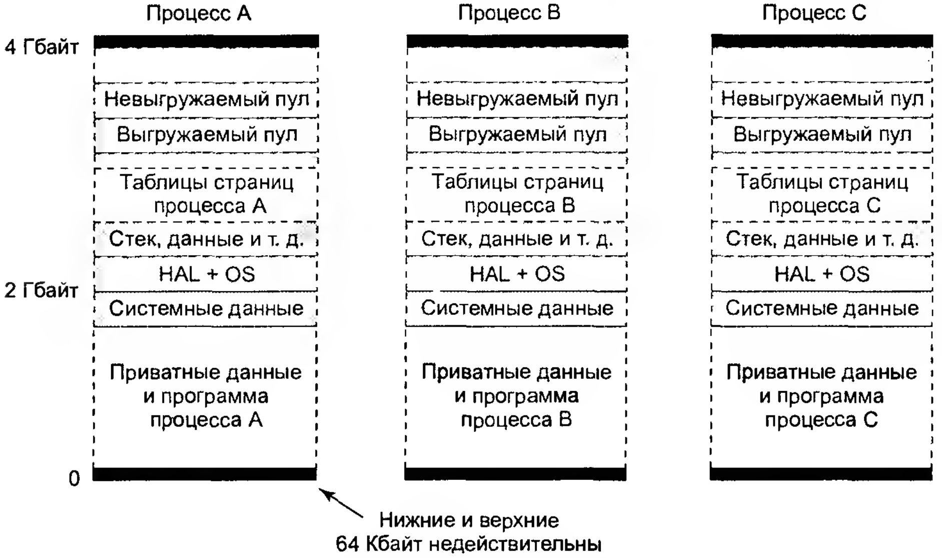


Рис. 11.11. Конфигурация виртуального адресного пространства для трех пользовательских процессов

Начиная с адреса 64 K, могут располагаться приватные данные и программа пользователя. Они могут занимать почти 2 Гбайт. Последний фрагмент этих 2 Гбайт памяти содержит некоторые системные указатели и таймеры, используемые совместно всеми пользователями в режиме доступа «только чтение». Отображение этих данных в эту область памяти позволяет всем процессам получать к ним доступ без лишних системных вызовов.

Верхние 2 Гбайт виртуального адресного пространства содержат операционную систему, включая код, данные и выгружаемый и невыгружаемый пулы (используемые для объектов и т. д.). Верхние 2 Гбайт используются совместно всеми процессами, кроме таблиц страниц, которые являются индивидуальными для каждого процесса. Верхние 2 Гбайт процессам в режиме пользователя запрещены для записи, а по большей части также запрещены и для чтения. Причина, по которой они размещаются здесь, заключается в том, что когда поток обращается к системному вызову, он переключается в режим ядра, но остается все тем же потоком. Если сделать всю операционную систему и все ее структуры данных (как и весь пользовательский процесс) видимыми в адресном пространстве потока, когда он переключается в режим ядра, то отпадает необходимость в изменении карты памяти или выгрузке кэша при входе в ядро. Все, что нужно сделать, — это переключиться на стек режима ядра. Платой за более быстрые системные вызовы при данном подхо-

де является уменьшение приватного адресного пространства для каждого процесса. Большим базам данных уже сейчас становится тесно в таких рамках, вот почему в версиях Windows 2000 Advanced server и Datacenter Server есть возможность использования 3 Гбайт для адресного пространства пользовательских процессов.

Каждая виртуальная страница может находиться в одном из трех состояний: свободном, зарезервированном и фиксированном. **Свободная страница** не используется в настоящий момент, и ссылка на нее вызывает страничное прерывание. Когда процесс запускается, все его страницы находятся в свободном состоянии, пока программа и исходные данные не будут отображены на их адресное пространство. Как только данные или программа отображаются на страницу, страница называется **фиксированной**. Обращение к фиксированной странице преобразуется при помощи аппаратного обеспечения виртуальной памяти и завершается успехом, если эта страница находится в оперативной памяти. В противном случае происходит страничное прерывание, операционная система находит требуемую страницу на диске и считывает ее в оперативную память.

Виртуальная страница может также находиться в **зарезервированном** состоянии, в таком случае эта страница не может отображаться, пока резервирование не будет явно удалено. Например, когда создается новый поток, в виртуальном адресном пространстве резервируется 1 Мбайт пространства для стека, но фиксируется только одна страница. Такая техника означает, что стек может вырасти до 1 Мбайт без опасения, что какой-либо другой поток захватит часть необходимого непрерывного виртуального адресного пространства. Помимо состояния (свободная, зарезервированная или фиксированная), у страниц есть также и другие атрибуты, например страница может быть доступной для чтения, записи или исполнения.

При выделении фиксированным страницам места резервного хранения используется интересный компромисс. Простая стратегия в данном случае состояла бы в отведении для каждой фиксированной страницы одной страницы в файле подкачки во время фиксации страницы. Это означало бы, что всегда есть место, куда записать каждую фиксированную страницу, если потребуется удалить ее из памяти. Недостаток такой стратегии заключается в том, что при этом может потребоваться файл подкачки размером со всю виртуальную память всех процессов. На большой системе, которой редко требуется выгрузка виртуальной памяти на диск, такой подход приведет к излишнему расходованию дискового пространства.

Чтобы не тратить пространство на диске понапрасну, в Windows 2000 фиксированным страницам, у которых нет естественного места хранения на диске (например, страницам стека), не выделяются страницы на диске до тех пор, пока не настанет необходимость их выгрузки на диск. Такая схема усложняет систему, так как во время обработки страничного прерывания может понадобиться обращение к файлам, в которых хранится информация о соответствии страниц, а чтение этих файлов может вызвать дополнительные страничные прерывания. С другой стороны, для страниц, которые никогда не выгружаются, пространства на диске не требуется.

Подобный выбор (усложнение системы или увеличение производительности и дополнительные функции), как правило, разрешается в пользу последнего, так как достоинства лучшей производительности и большего числа функций очевидны, тогда как недостатки усложнения системы (сложность поддержки и увеличение частоты сбоев) бывает сложно учесть. У свободных и зарезервированных стра-

ниц никогда не бывает теневых страниц на диске и обращение к ним всегда приводит к страничным прерываниям.

Теневые страницы на диске организованы в один или несколько файлов подкачки. Может быть организовано до 16 файлов подкачки, для повышения производительности операций ввода-вывода они могут быть распределены по отдельным дискам, которых также может быть до 16. У каждого файла есть начальный размер и максимальный размер, до которого он может вырасти при необходимости. Эти файлы могут сразу быть созданы максимального размера во время установки системы, чтобы уменьшить вероятность их сильной фрагментации, но с помощью панели управления позднее можно создать новые файлы. Операционная система следит за тем, какие виртуальные страницы на какую часть файла подкачки отображаются. Страницы, содержащие исполняемый текст программ, не дублируются в файлах подкачки. В файлах подкачки хранятся только изменяемые страницы.

В Windows 2000, как и во многих версиях UNIX, файлы могут отображаться напрямую на области виртуального адресного пространства (то есть занимать множество соседних страниц). После того как файл отображен на адресное пространство, он может читаться и писаться при помощи обычных команд обращения к памяти. Отображаемые на память файлы реализуются тем же способом, что и фиксированные страницы, но теневые страницы хранятся не в файле подкачки, а в файле пользователя. Поэтому при отображении файла на память версия файла, находящаяся в памяти, может отличаться от дисковой версии (вследствие записи в виртуальное адресное пространство). Однако когда отображение файла прекращается или файл принудительно выгружается на диск, дисковая версия снова приводится в соответствие с последними изменениями файла в памяти.

В Windows 2000 два и более процессов могут одновременно отображать на свои виртуальные адресные пространства, возможно, в различные адреса, одну и ту же часть одного и того же файла, как показано на рис. 11.12. (Файл *lib.dll* на рисунке отображен одновременно на два адресных пространства.) Читая и записывая слова памяти, процессы могут общаться друг с другом и передавать друг другу информацию с очень большой скоростью, так как копирование при этом не требуется. У различных процессов могут быть различные права доступа. Поскольку все процессы, использующие отображаемый на память файл, совместно используют одни и те же страницы, изменения, произведенные одним процессом, немедленно становятся видимыми для всех остальных процессов, даже если файл на диске еще не был обновлен. Также предпринимаются меры, благодаря которым процесс, открывающий файл для нормального чтения, видит текущие страницы в ОЗУ, а не устаревшие страницы с диска.

Следует отметить, что при совместном использовании двумя программами одного файла DLL может возникнуть проблема, если одна из программ изменит статические данные файла. Если не предпринять специальных действий, то другой процесс увидит измененные данные, что, скорее всего, не соответствует намерениям этого процесса. Эта проблема решается таким способом: все отображаемые страницы помечаются как доступные только для чтения, хотя в то же время некоторые из них тайно помечаются как в действительности доступные и для записи. Когда

к такой странице происходит обращение операции записи, создается приватная копия этой страницы и отображается на память. Теперь в эту страницу можно писать, не опасаясь задеть других пользователей или оригинальную копию на диске. Такая техника называется **копированием при записи**.

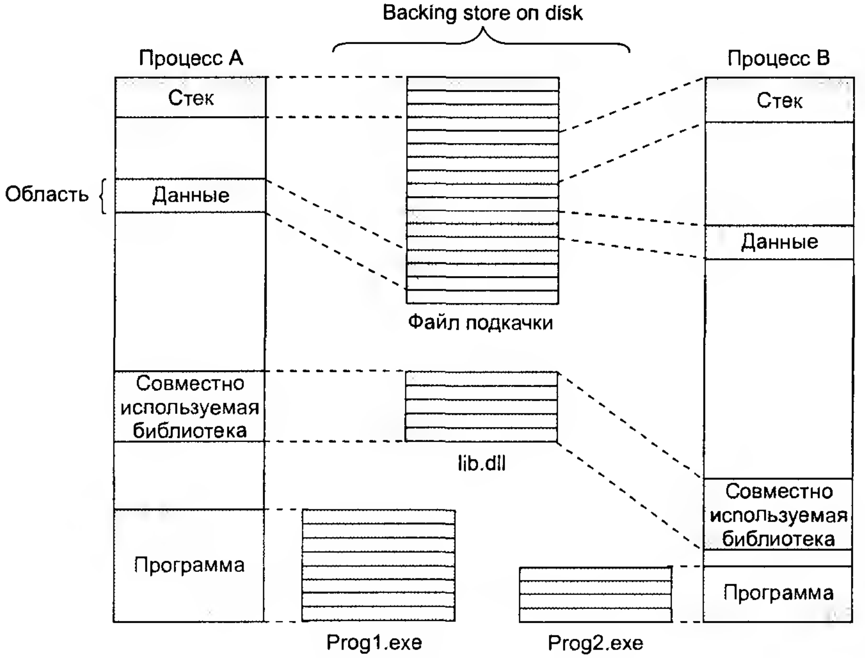


Рис. 11.12. Отображаемые на память области с их теневыми страницами на диске

Следует отметить, что и в случае, если текст программы отображается на два адресных пространства по различным адресам, также возникают проблемы с адресацией. Что будет, если первой командой будет JMP 300? Если процесс 1 отобразит эту программу на адрес 65 536, программа легко может быть настроена на новый адрес заменой этой команды командой JMP 65836. Но что будет, если второй процесс отобразит эту программу с адреса 131 072? Вместо передачи управления по адресу 131 372 команда JMP 65836 передаст его по адресу 65 836, после чего вся программа будет работать неверно. Решение заключается в использовании в совместно используемой программе только относительных смещений вместо абсолютных виртуальных адресов. К счастью, у большинства компьютеров есть команды, использующие относительную адресацию, а также команды, использующие абсолютную адресацию. Компиляторы могут использовать относительную адресацию, но они должны знать заранее, какой тип адресации использовать. Как правило, относительная адресация не используется постоянно, так как при этом снижается эффективность программы. Тип используемой адресации обычно задается ключом компиляции. Техника создания участка программы, который может работать независимо от адреса без настройки, называется **PIC** (Position Independent Code — позиционно-независимый код).

Несколько лет назад, когда 16-разрядные (или 20-разрядные) виртуальные адреса были стандартом, но у машин были мегабайты физической памяти, придумывались самые разнообразные трюки, позволяющие программам использовать больше физической памяти, чем помещалось в адресное пространство. Часто применялось так называемое **переключение банков** памяти, заключавшееся в том, что программа заменяла один физический блок памяти в своем адресном пространстве другим блоком. Когда появились 32-разрядные машины, программисты думали, что теперь у них никогда не будет недостатка в адресном пространстве. Они ошибались. Эта проблема снова возвращается. Большим программам часто бывает нужно больше, чем 2 или 3 Гбайт пользовательского адресного пространства, предоставляемого им системой Windows 2000. Таким образом, переключение банков памяти возвращается под названием **оконных расширений адреса**. Это свойство позволяет программам отображать в адресное пространство пользователя то одну, то другую большую область памяти (особенно за пределами 4-гигабайтной границы). Поскольку это имеет смысл только на серверах, у которых более 2 Гбайт физической памяти, мы отложим обсуждение этой проблемы до выхода следующего издания данной книги (к тому времени даже настольные компьютеры будут ощущать ограничение 32-разрядных адресов).

Системные вызовы управления памятью

Интерфейс Win32 API содержит множество функций, позволяющих процессу явно управлять своей виртуальной памятью. Наиболее важные функции перечислены в табл. 11.13. Все они работают с областью, состоящей либо из одной страницы, или с двумя или более страницами, располагающимися последовательно в виртуальном адресном пространстве.

Таблица 11.13. Основные функции Win32 API для управления виртуальной памятью в Windows 2000

Функция	Описание
VirtualAlloc	Зарезервировать или фиксировать область
VirtualFree	Освободить область или отменить фиксацию области
VirtualProtect	Изменить режим доступа (чтение/запись/выполнение) к области
VirtualQuery	Узнать состояние области
VirtualLock	Сделать область резидентной в памяти (то есть запретить ее выгрузку)
VirtualUnlock	Разрешить выгрузку области
CreateFileMapping	Создать объект отображаемого файла и (по желанию) присвоить ему имя
MapViewOfFile	Отобразить файл (часть файла) на адресное пространство
UnmapViewOfFile	Удалить отображаемый файл из адресного пространства
OpenFileMapping	Открыть созданный ранее объект отображаемого файла

Первые четыре функции API используются для выделения и освобождения областей виртуального адресного пространства, изменения их режима защиты и получения информации о текущем режиме. Выделенные области всегда начинаются с 64-килобайтных границ, что сводит к минимуму проблемы переноса систе-

мы на компьютеры будущего с большим размером страниц (до 64 Кбайт), чем у нынешних машин. Количество выделенного адресного пространства может быть меньше, чем 64 Кбайт, но оно должно состоять из целого числа страниц. Следующие две функции API дают процессу возможность зафиксировать страницы в памяти, запрещая их выгрузку, и отменить их фиксацию. Такая возможность может быть полезной, например, для программы реального времени. Операционной системой накладывается предел на количество фиксируемых страниц. На самом деле фиксированные страницы могут быть удалены из памяти, но только в том случае, когда весь процесс выгружается на диск. Когда процесс снова загружается в память, все его зафиксированные страницы также загружаются, прежде чем поток получает управление. Хотя они и не показаны в табл. 11.13, в операционной системе Windows 2000 также есть функции API, позволяющие процессу получать доступ к виртуальной памяти других процессов, которыми он может управлять (то есть тех, от которых у него есть дескриптор).

Последние четыре функции API, перечисленные в таблице, управляют отображением файлов на адресное пространство памяти. Чтобы отобразить файл на адресное пространство памяти, сначала следует создать объект отображения (см. табл. 11.6) при помощи функции `CreateFileMapping`. Эта функция возвращает дескриптор объекта отображения, а также может ввести имя объекта в файловую систему, чтобы другие процессы могли пользоваться этим объектом. Следующие две функции включают и выключают отображение файла на адресное пространство памяти. Последняя функция в таблице может использоваться процессом для отображения на память файла, уже использующегося подобным образом другим процессом. Таким образом, несколько процессов могут совместно использовать области своих виртуальных адресных пространств. Эта техника позволяет им записывать данные в ограниченные области памяти других процессов.

Реализация управления памятью

В операционной системе Windows 2000 поддерживается подгружаемое по требованию одинарное линейное 4-гигабайтное адресное пространство для каждого процесса. Сегментация в любой форме не поддерживается. Теоретически размер страниц может быть любой степенью двух, вплоть до 64 Кбайт. На компьютерах с процессором Pentium страницы имеют фиксированный размер в 4 Кбайт. На компьютерах с процессором Itanium они могут быть 8 или 16 Кбайт. Кроме того, сама операционная система может использовать страницы по 4 Мбайт, чтобы снизить размеры таблицы страниц.

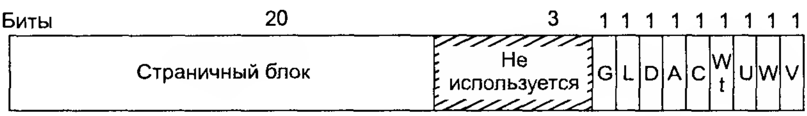
В отличие от планировщика, выбирающего отдельные потоки для запуска и не заботящегося о процессах, менеджер памяти занимается исключительно процессами и не беспокоится о потоках. В конце концов, именно процессы, а не потоки владеют адресным пространством, которым занимается менеджер памяти. При выделении области виртуального адресного пространства (на рис. 11.12 процессу *A* было выделено четыре области) менеджер памяти создает для нее **описатель виртуальной памяти** (VAD, Virtual Address Descriptor), в котором хранится информация о диапазоне отображаемых адресов, файле резервного хранения и смещении в файле для отображаемой части файла, а также режим доступа. Когда происходит

обращение к первой странице, создается каталог таблиц страниц, а указатель на нее помещается в описатель виртуальной памяти. Адресное пространство полностью описывается списком своих описателей виртуальной памяти. Такая схема позволяет поддерживать несплошные адресные пространства, так как неиспользуемые области между отображаемыми областями не потребляют ресурсов.

Обработка страничных прерываний

В операционной системе Windows 2000 опережающая подкачка страниц не используется ни в каком виде. Когда запускается процесс, в памяти не находится ни одной страницы процесса. При каждом страничном прерывании происходит передача управления ядру (которое понимается так, как изображено на рис. 11.2). Ядро формирует машинно-независимый описатель, в который помещается информация о том, что случилось, и передает его части исполняющей системы, выполняющей функции менеджера памяти. Менеджер памяти проверяет полученный описатель на корректность. Если страница, вызвавшая прерывание, попадает в фиксированную или зарезервированную область, он ищет адрес в списке описателей виртуальной памяти, находит (или создает) таблицу страниц и ищет в ней соответствующий элемент.

Элементы таблицы страниц различаются в разных архитектурах. Для компьютеров с процессором Pentium элемент таблицы для отображаемой страницы показан на рис. 11.13. У неотображаемых страниц также есть записи в таблице, но их формат несколько отличается. Например, если неотображаемая страница должна быть обнулена перед употреблением, этот факт отражается в таблице.



G: Глобальная страница для всех процессов
L: Большая (4 Мбайт) страница
D: Страница «грязная»
A: К странице были обращения
C: Кэширование разрешено/запрещено
Wt: Сквозная запись (без кэширования)
U: К странице возможен доступ в режиме пользователя
W: Запись в страницу разрешена
V: Действительная запись таблицы

Рис. 11.13. Запись таблицы для отображаемой страницы на компьютере с процессором Pentium

Для алгоритма подкачки наиболее важными битами записи таблицы страниц являются биты *A* и *D*. Они устанавливаются аппаратно и позволяют отслеживать наличие обращений к странице и записи в нее с момента последнего сброса этих битов. Страничные прерывания подразделяются на пять категорий:

1. Страница, к которой было обращение, не является фиксированной.
2. Произошло нарушение защиты.
3. Запись в совместно используемую страницу.
4. Стеку требуется дополнительная память.
5. Страница, к которой было обращение, является фиксированной, но в настоящий момент она не загружена в память.

Первый и второй случаи представляют собой фатальные ошибки, которые не могут быть исправлены или проигнорированы. У третьего случая симптомы схожи со вторым (попытка записи в страницу, для которой разрешено только чтение), но лечение этого случая возможно. В этом случае страница копируется в новый физический страничный блок, после чего для копии разрешается чтение/запись. Таким образом, работает копирование при записи. (Если совместно используемая страница помечена как доступная для записи во всех процессах, использующих ее, страничного прерывания при записи в такую страницу не возникает и копии при записи не возникает также.) В четвертом случае требуется выделение нового страничного блока и его отображение. Однако правила безопасности требуют, чтобы эта страница содержала только нули, что не позволяет новому процессу узнать, чем занимался предыдущий владелец страницы. Таким образом, нужно найти страницу, содержащую одни нули или, если это невозможно, нужно выделить другой страничный блок и обнулить его на месте. Наконец, пятый случай представляет собой нормальное страничное прерывание. Менеджер памяти находит страницу на диске и считывает ее в память.

Фактический механизм получения и отображения страниц весьма стандартен, поэтому мы не станем обсуждать здесь этот вопрос. Следует только отметить, что операционная система Windows 2000 не читает отдельные страницы прямо с диска. Вместо этого считывается несколько последовательных страниц, как правило, от 1 до 8, чтобы минимизировать количество обращений к диску. Для страниц, содержащих код программы, используются серии из большего числа страниц, чем при считывании страниц данных.

Алгоритм замещения страниц

Замена страниц происходит следующим образом. Система пытается поддерживать определенное количество свободных страниц в памяти, чтобы, когда произойдет страничное прерывание, свободная страница могла быть найдена немедленно, без необходимости сначала записать несколько других страниц на диск. В результате применения такой стратегии большинство страничных прерываний удовлетворяются при помощи всего одной дисковой операции (чтения страницы с диска), хотя иногда приходится выполнять две операции (запись на диск «грязной» страницы, после чего с диска читается требуемая страница).

Конечно, страницы, пополняющие список свободных страниц, должны откуда-то поступать. Поэтому настоящая работа алгоритма замещения страниц характеризуется тем, как эти страницы забираются у процессов и помещаются в список свободных страниц (в действительности существует четыре списка свободных страниц, но в данный момент для простоты будем считать, что это один список; детали мы обсудим позднее). Посмотрим теперь, как операционная система Windows 2000 освобождает страницы. Начнем с того, что в системе подкачки активно используется понятие рабочего набора. У каждого процесса (не у каждого потока) есть рабочий набор. Этот набор состоит из отображенных страниц, находящихся в памяти, при обращении к которым, следовательно, не происходит страничных прерываний. Размер и состав рабочего набора, естественно, меняются по мере работы процесса.

Рабочий набор каждого процесса описывается двумя параметрами: минимальным и максимальным размерами. Эти размеры не являются жесткими границами.

Процесс может иметь в памяти меньше страниц, чем значение нижней границы, или (при определенных обстоятельствах) больше установленного максимума. Вначале эти границы одинаковы для каждого процесса, но они могут меняться со временем. Начальное значение минимума по умолчанию находится в диапазоне от 20 до 50 страниц, а начальное значение максимума по умолчанию находится в диапазоне от 45 до 345 страниц, в зависимости от общего объема оперативной памяти. Значения по умолчанию могут быть изменены системным администратором.

Если происходит страничное прерывание, а размер рабочего набора меньше минимального значения, то к рабочему набору добавляется страница. С другой стороны, если происходит страничное прерывание, а размер рабочего набора больше максимального значения, то из рабочего набора (но не из памяти) изымается страница, чтобы выделить место для новой страницы. Этот алгоритм означает, что в операционной системе Windows 2000 используется локальный алгоритм, не позволяющий процессу получить слишком много памяти, что предотвращает причинение процессами ущерба друг другу. Однако система пытается настроить эти параметры. Например, если она замечает, что один процесс слишком активно занимается подкачкой (а остальные процессы нет), система может увеличить значение максимального предела для рабочего набора; таким образом, алгоритм представляет собой смесь локальных и глобальных решений. Тем не менее существует абсолютный предел размера рабочего набора: даже если в системе работает всего один процесс, он не может занять последние 512 страниц, чтобы оставить немного оперативной памяти для новых процессов.

Однако история на этом не заканчивается. Раз в секунду выделенный демон-поток ядра, называемый **менеджером балансового множества**, проверяет, достаточно ли в системе свободных страниц. Если свободных страниц меньше, чем нужно, он запускает **менеджер рабочих наборов**, который исследует рабочие наборы и освобождает дополнительные страницы. Менеджер рабочих наборов сначала определяет порядок, в котором нужно исследовать процессы. В первую очередь страницы отнимаются у больших процессов, которые бездействовали в течение долгого времени. В последнюю очередь рассматривается процесс переднего плана.

Затем менеджер рабочих наборов начинает исследование процессов в выбранном порядке. Если рабочий набор процесса в настоящий момент оказывается меньше своего нижнего предела или с момента последней инспекции число страничных прерываний у этого процесса было выше определенного уровня, то страницы у него не отнимаются. В противном случае менеджер рабочих наборов отнимает у процесса одну или несколько страниц. Количество забираемых у процесса страниц довольно сложным образом зависит от общего объема ОЗУ, от того, насколько много требуется памяти текущим процессам, от того, как размер текущего рабочего набора соотносится с верхним и нижним пределами, а также от других параметров. Все страницы рассматриваются по очереди.

На однопроцессорной машине если бит обращений к странице сброшен, то счетчик, связанный со страницей, увеличивается на единицу. Если этот бит установлен в единицу, счетчик обнуляется. После сканирования из рабочего набора удаляются страницы с наибольшими значениями счетчика. Поток продолжает изучать процессы, пока он не высвободит достаточного количества страниц, после чего он останавливается. Если полный перебор всех процессов не привел к освобож-

дению достаточного числа страниц, менеджер рабочих наборов начинает второй проход, на котором он уже «состригает» с процессов страницы более агрессивно, даже отнимая их при необходимости у процессов, размер рабочего набора которых меньше минимального.

На многопроцессорной системе алгоритм, основанный на проверке бита обращений, уже не работает, так как хотя текущий центральный процессор не обращался в последнее время к данной странице, к ней могли обращаться другие центральные процессоры. Исследование же битов обращений всех центральных процессоров представляет собой слишком дорогое удовольствие. Поэтому бит обращений вообще не учитывается, а удаляются самые старые страницы.

Следует отметить, что с точки зрения процедуры замены страниц операционная система сама рассматривается как процесс. Она владеет страницами и у нее также есть рабочий набор. Этот рабочий набор тоже может быть уменьшен. Однако некоторые части системы и невыгружаемый пул фиксированы в памяти и не могут выгружаться ни при каких обстоятельствах.

Управление физической памятью

Выше упоминалось, что в действительности в операционной системе Windows 2000 свободные страницы учитываются в четырех списках. Теперь настала пора познакомиться с ними поближе. Каждая страница памяти находится в одном или нескольких рабочих наборах или в одном из этих четырех списков, показанных на рис. 11.14. В списке чистых (резервных) и в списке грязных (модифицированных) страниц учитываются страницы, которые недавно были удалены из рабочих наборов, но все еще находятся в памяти и все еще ассоциированы с процессами, использовавшими их. Различие между ними заключается в том, что у чистых страниц есть копия на диске, тогда как у модифицированных страниц таких копий нет, и эти страницы еще предстоит сохранить. В список свободных страниц входят чистые страницы, уже не ассоциированные ни с какими процессами. В список обнуленных страниц входят страницы, не ассоциированные ни с какими процессами и заполненные нулями. Пятый список состоит из физически дефектных страниц памяти. Это гарантирует, что эти страницы ни для чего не используются.

Страницы перемещаются между рабочими наборами и различными списками менеджером рабочих наборов и другими потоками-демонами ядра. Рассмотрим эти переходы. Когда менеджер рабочих наборов удаляет страницу из рабочего набора, страница попадает на дно списка чистых страниц или списка модифицированных страниц в зависимости от своего состояния. Этот переход показан на рисунке как (1). В обоих списках хранятся действительные страницы, поэтому если происходит страничное прерывание и требуется одна из этих страниц, она удаляется из списка и возвращается в свой рабочий набор без операции дискового ввода-вывода (2). Когда процесс завершает свою работу, то все его страницы, которые не используются другими процессами, попадают в список свободных страниц (3). Эти страницы уже не ассоциированы с каким-либо процессом и не могут возвращаться в рабочие наборы по страничному прерыванию.

Другие переходы вызываются другими демонами. Раз в 4 с запускается **поток свопера** в поисках процесса, все потоки которого бездействовали в течение определенного интервала времени. Если ему удается найти такие процессы, он от-

крепляет стеки этих процессов и перемещает страницы процессов в списки чистых и грязных страниц (1).

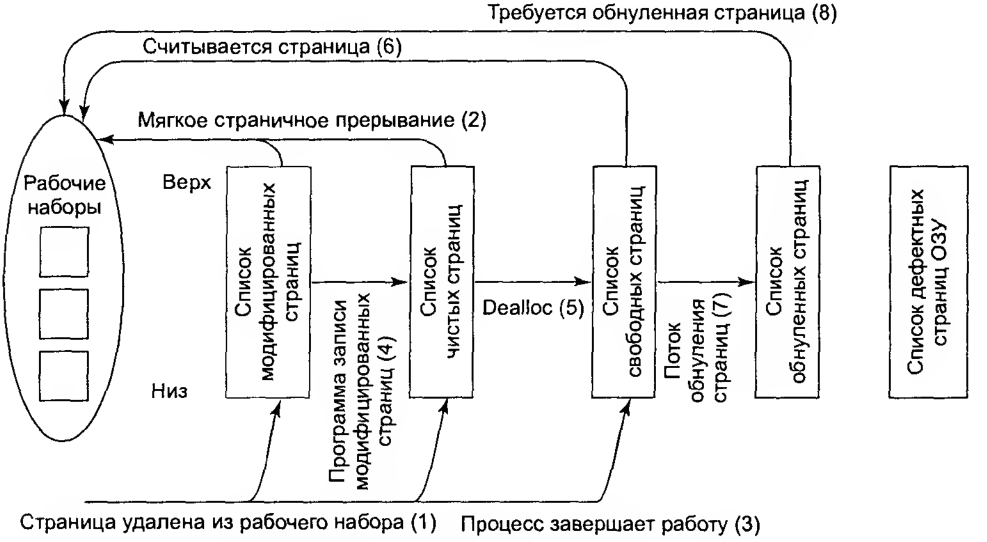


Рис. 11.14. Списки страниц и переходы между ними

Два других демона, **демон записи отображенных страниц** и **демон записи модифицированных страниц**, просыпаются время от времени, чтобы проверить, достаточно ли чистых страниц. Если количество чистых страниц ниже определенного уровня, они берут страницы из верхней части списка модифицированных страниц, записывают их на диск, а затем помещают их в список чистых страниц (4). Первый демон занимается записью в отображаемые файлы, а второй пишет страницы в файлы подкачки. В результате их деятельности грязные страницы становятся чистыми.

Причина наличия двух демонов, занимающихся очисткой страниц, заключается в том, что отображаемый на память файл может вырасти в результате записи в него. При этом росте потребуются новые свободные блоки диска. Отсутствие в памяти свободного места для записи в него страниц может привести к взаимоблокировке. Второй поток может вывести ситуацию из тупика, записывая страницы в файл подкачки, который никогда не увеличивается в размерах. Никто и не говорил, что операционная система Windows 2000 проста.

Обсудим другие переходы на рис. 11.14. Если процесс освобождает страницу, эта страница более не связана с процессом и может быть помещена в список свободных страниц (5), если только она не используется совместно другими процессами. Когда страничное прерывание требует страничный блок, чтобы поместить в него страницу, которая должна быть считана, этот блок по возможности берется из списка свободных страниц (6). Не имеет значения, что эта страница может все еще содержать конфиденциальную информацию, так как вся она будет тут же целиком перезаписана. При увеличении стека ситуация складывается иная. В этом случае требуется пустой страничный блок и правила безопасности требуют, чтобы стра-

ница содержала все нули. По этой причине другой демон ядра, **поток обнуления страниц**, работает с минимальным приоритетом (см. рис. 11.8), стирая содержимое страниц в списке свободных страниц и помещая их в список обнуленных страниц (7). Когда центральный процессор простаивает и в списке свободных страниц есть страницы, поток обнуления страниц может обнулять их, так как обнуленная страница более полезна, чем просто свободная страница.

Наличие всех этих списков приводит к необходимости принятия некоторых стратегических решений. Например, предположим, что страница должна быть считана с диска, а список свободных страниц пуст. Теперь система вынуждена выбирать между чистыми страницами из списка резервных страниц (которые могут потребоваться при очередном страничном прерывании) и пустыми страницами из списка обнуленных страниц (в результате работа по обнулению страниц окажется выполненной впустую). Что лучше? Если центральный процессор ничем не занят, а обнуляющий страницы поток запускается часто, лучше взять обнуленную страницу, так как в них нет недостатка. Однако если центральный процессор всегда занят, а диск по большей части простаивает, лучше взять страницу из списка резервных страниц, чтобы избежать излишних затрат процессорного времени на обнуление еще одной страницы, когда вырастет стек.

Еще одна загадка. Насколько агрессивно должны демоны перемещать страницы из списка грязных страниц в список чистых страниц? Лучше иметь большое количество чистых страниц, чем большое количество грязных страниц, так как чистую страницу можно использовать мгновенно. Однако агрессивная политика очистки страниц означает большее количество операций дискового ввода-вывода; кроме того, есть шанс, что только что очищенная страница снова будет затребована в рабочий набор страничным прерыванием и снова испачкана.

В операционной системе Windows 2000 конфликты подобного рода разрешаются при помощи сложных эвристических алгоритмов, угадывания, учета предыстории, правил большого пальца и настройки параметров, устанавливаемых системным администратором. Более того, эта программа настолько сложна, что разработчики не любят менять в ней что-либо из-за страха сломать в системе какой-нибудь фрагмент, структуру и назначение которого сегодня уже никто не понимает.

Для отслеживания всех страниц и всех списков операционная система Windows 2000 содержит базу данных страничных блоков, состоящую из записей по числу страниц ОЗУ (рис. 11.15). Эта таблица проиндексирована по номеру физического страничного блока. Записи таблицы имеют фиксированную длину, но для различных типов записей используются различные форматы (например, для действительных записей и для недействительных). Действительные записи содержат информацию о состоянии страницы, а также счетчик, хранящий число ссылок на эту страницу в таблицах страниц. Этот счетчик позволяет системе определить, когда страница уже более не используется. Если страница находится в рабочем наборе, то в записи также указывается номер рабочего набора. Кроме того, в записи содержится указатель на таблицу страниц, в которой есть указатель на эту страницу (если такая таблица страниц есть). Страницы, используемые совместно, учитываются особо. Также запись содержит ссылку на следующую страницу в списке (если такая есть) и различные другие поля и флаги, такие как «страница читается», «страница пишется» и т. д.

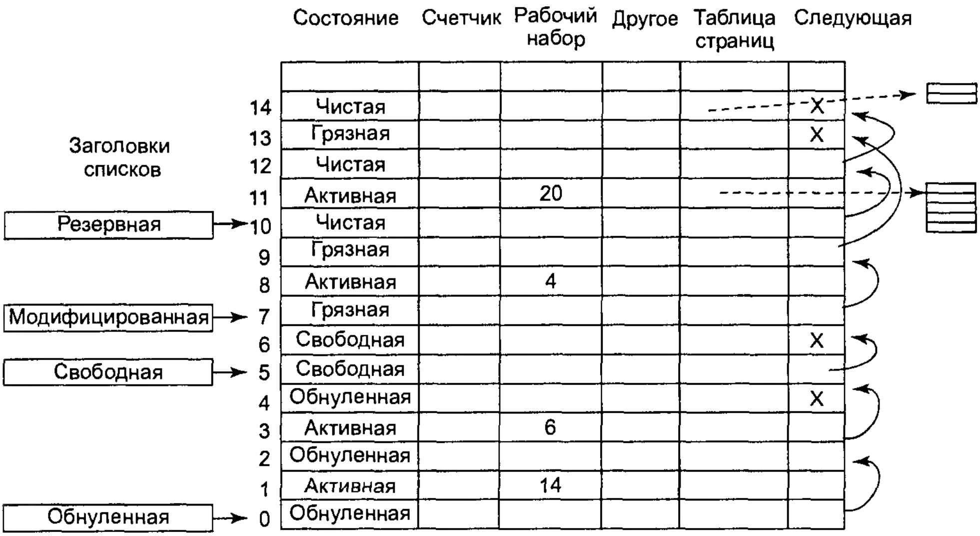


Рис. 11.15. Некоторые основные поля базы данных страничных блоков

Итак, управление памятью представляет собой очень сложную подсистему с большим количеством структур данных, алгоритмов и эвристических методов. Во многом она является саморегулируемой, но у нее есть также множество кнопок, на которые может нажимать системный администратор, чтобы влиять на производительность системы. Увидеть эти кнопки и связанные указатели можно при помощи различных программ, входящих в состав упоминавшихся ранее разнообразных наборов инструментальных средств. Вероятно, важнее всего здесь помнить, что управление памятью в реальных системах намного сложнее простого алгоритма подкачки, вроде алгоритма часов или алгоритма старения.

Ввод-вывод в Windows 2000

Задача системы ввода-вывода Windows 2000 заключается в предоставлении каркаса для эффективного управления широким спектром устройств ввода-вывода. К устройствам ввода относятся различные типы клавиатур, мышей, сенсорных панелей, джойстиков, сканеров, фотокамер, видеокамер, устройств чтения штрих-кода и микрофонов. Устройства вывода включают в себя мониторы, принтеры, плоттеры, бимеры, устройства записи CD-ROM и звуковые карты. К запоминающим устройствам относятся накопители на гибких дисках, жесткие диски IDE и SCSI, устройства чтения CD-ROM, устройства чтения DVD, накопители на Zip-дисках и магнитофоны. Наконец, существуют еще такие устройства, как часы, сети, телефоны и видеокамеры со встроенными накопителями. Без всякого сомнения, в ближайшие годы будут изобретены новые устройства, поэтому операционная система Windows 2000 была спроектирована таким образом, чтобы к системе ввода-вывода можно было легко добавлять новые устройства. В следующих разделах мы рассмотрим некоторые вопросы, относящиеся к вводу-выводу.

Основные понятия

Менеджер ввода-вывода родственен менеджеру plug-and-play. Основная идея механизма plug-and-play заключается в настраиваемой шине. За многие годы было разработано множество шин, включая PC Card, PCI, USB, IEEE 1394 и SCSI, поэтому менеджер plug-and-play может послать каждому разъему запрос и попросить устройство назвать себя. Определив, что за устройство подключено к шине, менеджер plug-and-play выделяет для него аппаратные ресурсы, такие как уровни прерываний, находит необходимые драйверы и загружает их в память. При загрузке каждого драйвера для него создается **объект драйвера**. Для некоторых шин, например SCSI, настройка происходит только во время загрузки операционной системы. Для других шин, таких как USB и IEEE 1394, она может производиться в любой момент, для чего требуется тесный контакт между менеджером plug-and-play, драйвером шины (который и выполняет настройку) и менеджером ввода-вывода.

Менеджер ввода-вывода также тесно связан с менеджером энергопотребления. Менеджер энергопотребления может перевести компьютер в одно из шести состояний, которые можно примерно описать следующим образом:

1. Полностью действующий.
2. Режим сниженного энергопотребления-1: мощность, потребляемая центральным процессором, снижается, ОЗУ и кэш работают, возможен мгновенный переход в режим полного действия.
3. Режим сниженного энергопотребления-2: центральный процессор и ОЗУ работают; кэш центрального процессора отключен; возможно продолжение работы с текущего значения счетчика команд.
4. Режим сниженного энергопотребления-3: центральный процессор и кэш отключены; ОЗУ работает; возможен перезапуск с фиксированного адреса.
5. Режим сниженного энергопотребления-3: центральный процессор и кэш отключены; ОЗУ работает; возможен перезапуск с фиксированного адреса.
6. «Зимняя спячка»: центральный процессор, кэш и ОЗУ отключены; возможен перезапуск из сохраненного на диске файла.
7. Выключен: все выключено; требуется полная перезагрузка.

Устройства ввода-вывода также могут находиться в различных состояниях. Включением и выключением этих устройств занимаются вместе менеджер энергопотребления и менеджер ввода-вывода. Обратите внимание, что состояния со 2 по 6 используются, только если центральный процессор бездействовал в течение определенного времени.

Это, может быть, кажется несколько удивительным, но формально все файловые системы представляют собой драйверы ввода-вывода. Обращения к блокам диска от пользовательских процессов сначала посылаются менеджеру кэша. Если менеджер кэша не может удовлетворить запрос из кэша, он просит менеджер ввода-вывода вызвать драйвер соответствующей файловой системы, чтобы тот получил требуемый блок с диска.

Интересная особенность операционной системы Windows 2000 заключается в поддержке **динамических дисков**. Эти диски могут распространяться на несколько дисковых разделов или даже физических дисков и их можно переконфигури-

ровать на лету, без перезагрузки. Таким образом, логические тома более не привязаны к отдельному разделу или даже одному жесткому диску, что позволяет получить файловую систему, располагающуюся на нескольких дисках прозрачным для пользователя способом.

Другой интересный аспект Windows 2000 заключается в поддержке асинхронного ввода-вывода. Поток может начать операцию ввода-вывода, а затем продолжить выполнение параллельно с вводом-выводом. Такая возможность особенно важна для серверов. Существует множество способов, с помощью которых поток может определить, что операция ввода-вывода завершена. Один из способов состоит в создании в момент обращения к вызову ввода-вывода объекта события, а потом ожидания этого события. Другой способ заключается в указании очереди, в которую будет послано сообщение о завершении операции ввода-вывода. Третий способ заключается в предоставлении процедуры обратного вызова, к которой обращается система, когда операция ввода-вывода завершена.

Вызовы ввода-вывода API

В операционной системе Windows 2000 есть более 100 вызовов API для работы с различными устройствами ввода-вывода, включая мыши, звуковые карты, телефоны, магнитофоны и т. д. Вероятно, наиболее важной является графическая система, для которой существует несколько тысяч вызовов Win32 API. В разделе «Программное обеспечение вывода для Windows» главы 5 мы начали обсуждение графической системы Windows 2000. Здесь мы продолжим обсуждение этой темы, рассмотрим несколько категорий интерфейса Win32 API, в каждую из которых входит множество вызовов. Краткий список категорий Win32 API приведен в табл. 11.14. Как упоминалось в главе 5, графическим функциям интерфейса Win32 API было посвящено множество 1500-страничных книг.

Таблица 11.14. Некоторые категории вызовов Win32 API

Группа API	Описание
Управление окнами	Создание, уничтожение окон, управление окнами
Меню	Создание, уничтожение и добавление пунктов меню
Диалоговые окна	Отображение диалоговых окон и сбор информации
Рисование и черчение	Отображение точек, линий и геометрических фигур
Текст	Вывод текста с использованием определенного шрифта, размера и цвета
Растровые изображения и значки	Отображение на экране растровых изображений и значков
Цвета и палитры	Управление набором доступных цветов
Буфер обмена	Передача информации от одного приложения другому
Ввод	Получение информации от мыши и клавиатуры

Существуют вызовы Win32 для создания и уничтожения окон, а также для управления окнами. Есть множество стилей и параметров, которые могут указываться при создании окна, такие как заголовки, рамки, цвета, размеры и полосы прокрутки. Окна могут быть фиксированные и перемещаемые, постоянного или

изменяемого размера. Существуют вызовы для получения и изменения свойств окон, а также для отправления сообщений окнам.

Многие окна содержат меню, поэтому существуют вызовы Win32 API для создания и удаления меню и строк меню. Динамические меню могут появляться и убираться с экрана. Пункты меню могут выделяться, затемняться и располагаться каскадом.

Диалоговые окна появляются, чтобы информировать пользователя о каком-либо событии или чтобы задать ему вопрос. Они могут содержать кнопки, ползунки и текстовые поля. С диалоговыми окнами также могут быть ассоциированы звуки, например для предупреждающих сообщений.

Существуют сотни функций для рисования и черчения, варьирующихся от задания одного пиксела до выполнения сложных операций с областями отсечения. Есть много вызовов для черчения линий и разнообразных замкнутых геометрических фигур с возможностью детального управления текстурами, цветом, шириной и многими другими атрибутами.

Другая группа вызовов относится к отображению текста. Вызов функции TextOut для вывода текста очень прост. Все сложности, связанные с выводом текста, заключаются в управлении цветом, размерами точек, гарнитурой шрифта, шириной символов, глифами, величиной межзнакового интервала и другими деталями. К счастью, преобразование текста в растр выполняется автоматически.

Растровые изображения представляют собой небольшие прямоугольные блоки пикселов, которые можно вывести на экран с помощью вызова BitBlt. Они используются для значков и иногда для текста. Существует множество вызовов для создания, удаления значков, а также для управления ими.

На многих дисплеях используется цветовой режим с использованием 256 или 65 536 из 2^{24} возможных цветов, что позволяет использовать для представления каждого пиксела всего один или два байта соответственно. В этих случаях требуется палитра цветов, чтобы указать, какие из 256 или 65 536 цветов доступны. Вызовы в этой группе позволяют создавать и удалять палитры, управлять ими, выбирать ближайший к заданному цвету доступный цвет, а также соблюдать соответствие цветов принтера цветам на экране.

Многие программы Windows 2000 позволяют пользователям выбирать некоторые данные (например, блок текста, фрагмент чертежа, набор ячеек в электронной таблице), помещать их в буфер обмена и вставлять данные оттуда в другие приложения. Определено множество форматов буфера обмена, включая текст, растровые изображения, объекты и метафайлы. Последние представляют собой множества вызовов Win32 рисования, что позволяет вырезать и копировать части рисунков. Данная группа вызовов Win32 позволяет помещать данные в буфер обмена, получать их из буфера обмена, а также управлять буфером обмена.

Наконец, существуют вызовы Win32, управляющие вводом данных. Среди этих вызовов нет вызовов для графических приложений, читающих входные данные с клавиатуры, так как графические приложения управляются событиями. Основная программа состоит из большого цикла, получающего входные сообщения. Когда пользователь вводит что-то интересное, программе посылается сообщение, содержащее введенную информацию. С другой стороны, существуют вызовы, относящиеся к мыши, такие как чтение координат (x, y) ее курсора и состояния кнопок. Некоторые вызовы ввода в действительности представляют собой вызовы выво-

да, например позволяющие выбрать изображение курсора мыши и управляющие перемещением курсора по экрану (эти действия представляют собой вывод на экран). Неграфические приложения могут читать с клавиатуры.

Реализация ввода-вывода

О графических вызовах интерфейса Win32 можно рассказывать бесконечно. Однако теперь пора взглянуть на то, как менеджер ввода-вывода реализует графические и другие функции ввода-вывода. Основная функция менеджера ввода-вывода заключается в создании каркаса, в котором могут работать различные устройства ввода-вывода. Основную структуру каркаса образуют набор независимых от устройств процедур для определенных аспектов ввода-вывода и набор загруженных драйверов для общения с устройствами.

Драйверы устройств

Чтобы гарантировать, что драйверы устройств хорошо работают с остальной частью системы Windows 2000, корпорация Microsoft определила для драйверов модель **Windows Driver Model**, которой драйверы устройств должны соответствовать. Более того, корпорация Microsoft также предоставляет набор инструментов, который должен помочь разработчикам драйверов в создании драйверов, соответствующих модели Windows Driver Model. В данном разделе мы кратко рассмотрим эту модель. Согласующиеся с ней драйверы должны удовлетворять всем следующим требованиям (а также некоторым другим):

1. Обрабатывать входящие запросы ввода-вывода, поступающие в стандартном формате.
2. Основываться на объектах, как и остальная часть системы Windows 2000.
3. Позволять динамическое добавление или удаление устройств plug-and-play.
4. Допускать, когда это возможно, управление энергопотреблением.
5. Допускать реконфигурацию в терминах использования ресурсов.
6. Быть реентерабельными для возможности их использования на мультипроцессорах.
7. Обладать переносимостью между системами Windows 98 и Windows 2000.

Запросы ввода-вывода передаются драйверам в виде стандартизированных пакетов, называемых **IRP** (Input/output Request Packet — пакет запроса ввода-вывода). Драйверы, согласующиеся с моделью Windows Driver Model, должны уметь обрабатывать пакеты IRP. Драйвер должен поддерживать работу с объектами, то есть поддерживать определенный список методов, к которым может обращаться остальная система. Он также должен корректно работать с другими объектами операционной системы Windows 2000, доступ к которым осуществляется при помощи дескрипторов объектов.

Драйверы, согласующиеся с моделью Windows Driver Model, должны полностью поддерживать устройства plug-and-play. Это означает, что если устройство, управляемое драйвером, внезапно добавляется в систему или удаляется из системы, драйвер должен быть готов к получению данной информации и корректной

реакции на эту информацию, даже в том случае, если устройство удаляется в момент обращения к нему. Также драйверы должны поддерживать управление энергопотреблением для тех устройств, для которых это возможно. Например, если система решает, что теперь пора перейти в режим низкого энергопотребления, все драйверы должны поддерживать этот режим, чтобы сберегать энергию. Они также должны поддерживать обратный переход в режим нормального функционирования.

Драйвер должен быть настраиваемым, что означает отсутствие каких бы то ни было встроенных предположений о линиях прерываний или портах ввода-вывода, используемых определенным устройством. Например, на компьютерах IBM PC и сменивших их моделях порт принтера более 20 лет имел адрес 0x378 и вряд ли будет изменен теперь. Но драйвер принтера, в который этот адрес жестко зашит, не является согласующимся с моделью Windows Driver Model.

Драйверы устройств также должны работать на мультипроцессорах, так как поддержка мультипроцессоров была заложена в операционную систему Windows 2000 при разработке. Это требование означает, это во время обработки драйвером запроса от одного центрального процессора может прийти запрос от другого центрального процессора. Второй центральный процессор может начать выполнение программы драйвера одновременно с первым центральным процессором. Драйвер должен функционировать корректно, даже когда он вызывается одновременно двумя и более центральными процессорами. Это означает, что доступ ко всем чувствительным структурам данных должен предоставляться только внутри критических областей. Простое предположение, что других обращений к драйверу не будет, пока не завершится обработка текущего обращения к нему, недопустимо.

Наконец, драйверы, согласующиеся с моделью Windows Driver Model, должны работать не только в операционной системе Windows 2000, но и в системе Windows 98. Однако может оказаться необходимой перекомпиляция драйвера на каждой системе и использование команд препроцессора С, чтобы устранить зависимость от платформы.

В операционной системе UNIX обращение к драйверам производится по номеру главного устройства. В Windows 2000 применяется другая схема. Во время загрузки операционной системы или в тот момент, когда в систему добавляется новое устройство plug-and-play, поддерживающее установку без перезагрузки системы, операционная система Windows 2000 автоматически обнаруживает его и вызывает менеджер plug-and-play. Менеджер plug-and-play запрашивает у устройства название фирмы-производителя и номер модели устройства. Вооружившись данными сведениями, он ищет драйвер для данного устройства в определенном каталоге на жестком диске. Если этого драйвера нет, он отображает диалоговое окно, в котором пользователю предлагается вставить гибкий диск или CD-ROM с драйвером. Когда драйвер обнаружен, он загружается в память.

Каждый драйвер должен поставлять набор процедур, которые могут быть вызваны для получения требуемого обслуживания. Первая процедура, называемая *DriverEntry*, инициализирует драйвер. Она вызывается сразу после загрузки драйвера. Процедура может создавать таблицы и структуры данных, но не должна обращаться к самому устройству. Она также заполняет некоторые поля объекта драйвера, созданного менеджером ввода-вывода при загрузке драйвера. Поля в объекте драйвера включают указатели на все остальные процедуры, предоставляемые драйвером.

Кроме того, для каждого устройства, управляемого драйвером (например, для каждого диска IDE, управляемого драйвером диска IDE), создается **объект устройства** и инициализируется так, чтобы он указывал на объект драйвера. Эти объекты драйверов помещаются в специальный каталог *\??*. При наличии объекта устройства можно легко найти объект драйвера и, таким образом, обращаться к его методам.

Вторая процедура драйвера называется *AddDevice*. Она вызывается (менеджером plug-and-play) всего один раз для каждого добавляемого устройства. После этого драйвер вызывается первым пакетом IRP, который устанавливает вектор прерываний и инициализирует аппаратуру. Кроме того, драйвер должен содержать процедуру обработки прерываний, различные процедуры, управляющие таймерами, путь быстрого ввода-вывода, управление DMA, позволять прервать исполняющийся текущий запрос и многое другое. В результате драйверы Windows 2000 настолько сложны, что этой теме посвящено множество книг [50, 252, 345].

В операционной системе Windows 2000 драйвер должен сам выполнять всю работу, как, например, выполняет ее драйвер принтера на рис. 11.16. С другой стороны, в системе Windows 2000 могут существовать стеки драйверов. Это означает, что запрос может проходить через целую последовательность драйверов, каждый из которых выполняет свою часть работы. Два таких драйвера также показаны на этом рисунке.

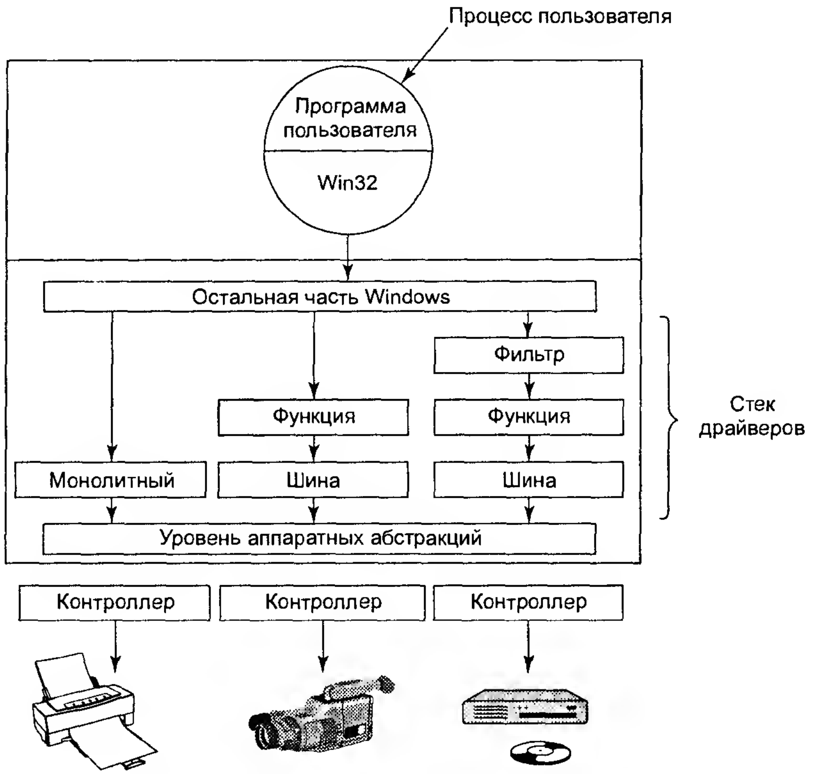


Рис. 11.16. Система Windows 2000 позволяет создавать драйверные стеки

Стеки драйверов позволяют отделить управление шиной от управления собственно устройством. Управление шиной PCI отличается большой сложностью, что вызвано большим количеством режимов и транзакций шины. Таким образом, отделение управления шиной от управления устройством, подключенным к данной шине, облегчает работу по созданию драйвера. Программисту, пишущему драйвер устройства, более не нужно изучать вопрос управления шиной. Он может просто использовать стандартный драйвер шины, находящийся в стеке драйверов. У драйверов USB и SCSI есть части, специфичные для конкретных устройств, и общая часть, для которой используются отдельные драйверы.

Кроме того, стеки драйверов позволяют добавлять в стек **драйверы-фильтры**. Фильтрующий драйвер выполняет некоторые преобразования проходящих через них данных. Например, фильтрующий драйвер может сжать данные по пути к диску или зашифровать их по дороге в сеть. Помещение драйверного фильтра в стек драйверов означает, что ни прикладная программа, ни настоящий драйвер устройства не должны знать о присутствии фильтрующего драйвера и что фильтрующий драйвер работает автоматически для всех данных, поступающих с устройства или на устройство.

Файловая система Windows 2000

Операционная система Windows 2000 поддерживает несколько файловых систем, самыми важными из которых являются **FAT-16**, **FAT-32** и **NTFS** (New Technology File System — файловая система новой технологии). Файловая система FAT-16 — это старая файловая система MS-DOS. В ней используются 16-разрядные дисковые адреса, что ограничивает размер дискового раздела двумя гигабайтами. В файловой системе FAT-32 используются 32-разрядные дисковые адреса и поддерживаются дисковые разделы размером до 2 Тбайт. Система NTFS представляет собой новую файловую систему, разработанную специально для Windows NT и перенесенную в Windows 2000. В ней используются 64-разрядные дисковые адреса, таким образом, теоретически эта файловая система может поддерживать дисковые разделы размером до 2^{64} байт, хотя по другим техническим причинам их размер ограничен меньшими размерами. Операционной системой Windows 2000 также поддерживаются файловые системы для CD-ROM и DVD, в которых разрешено только чтение. Одна и та же работающая операционная система может одновременно иметь доступ к нескольким файловым системам различного типа.

В данной главе мы обсудим файловую систему NTFS, так как это современная файловая система, не обремененная необходимостью полной совместимости с файловой системой MS-DOS, основанной на файловой системе CP/M, которая разрабатывалась для 8-дюймовых гибких дисков более 20 лет назад. Времена изменились, и 8-дюймовые гибкие диски уже не считаются современными. Устарели и файловые системы для этих дисков. Кроме того, файловая система NTFS во многом отличается от файловой системы UNIX как интерфейсом пользователя, так и в плане реализации, что делает файловую систему NTFS хорошим примером для изучения.

Файловая система NTFS является большой и сложной системой, и наша ограниченность в пространстве не позволяет нам обсудить все аспекты этой системы. Тем не менее приведенный ниже материал должен дать достаточное впечатление о ней.

Основные понятия

Длина имени файла в системе NTFS ограничена 255 символами; полная длина пути ограничивается 32 767 символами. Для имен файлов используется кодировка Unicode, что позволяет пользователям в странах, в которых не используется латинский алфавит (например, в Греции, Японии, Индии, России и Израиле), писать имена файлов на своем родном языке. Так, φιλε представляет собой вполне допустимое имя файла. Файловая система NTFS полностью поддерживает имена, чувствительные к регистру (таким образом, *foo* отличается от *Foo* и *FOO*). К сожалению, интерфейсом Win32 API не полностью поддерживается чувствительность к регистру для имен файлов и совсем не поддерживается для имен каталогов, поэтому это преимущество теряется при обращении к программам, обязанным использовать интерфейс Win32 (например, для совместимости с Windows 98).

Файл в системе NTFS — это не просто линейная последовательность байтов, как файлы в системах FAT-32 и UNIX. Вместо этого файл состоит из множества атрибутов, каждый из которых представляется в виде потока байтов. Большинство файлов имеет несколько коротких потоков, таких как имя файла и его 64-битовый идентификатор, плюс один длинный (неименованный) поток с данными. Однако у файла может быть и несколько длинных потоков данных. При обращении к каждому потоку после имени файла через двоеточие указывается имя потока, например *foo:stream1*. У каждого потока своя длина. Каждый поток может блокироваться независимо от остальных потоков. Идея нескольких потоков позаимствована у системы Apple Macintosh, в которой файлы имеют по два потока, ветвь данных и ветвь ресурса. Эта концепция была инкорпорирована в файловую систему NTFS, чтобы сервер с системой NTFS мог обслуживать клиенты Macintosh.

Файловые потоки могут использоваться не только для совместимости с Macintosh. Например, программа редактирования фотографий может использовать неименованный поток для основного изображения, а именованный поток — для небольшой пиктограммы. Эта схема проще, чем традиционный способ, при котором изображения помещаются в один и тот же файл, одно за другим. Другой пример использования потоков данных — электронная обработка текста. Эти программы часто создают две версии документа, временную для использования во время редактирования и окончательную версию, когда пользователь закончил работу. Если поместить временную версию в именованный поток, а окончательную версию в неименованный поток, обе версии автоматически оказываются в одном файле и без какой-либо дополнительной обработки пользуются одинаковыми правами доступа, временными штампами и т. д.

Максимальная длина потока составляет 2^{64} байт. Чтобы получить представление о том, насколько велик поток в 2^{64} байт, представьте, что поток записан в двоичном виде, где каждый символ 0 и 1 занимает 1 мм. В этом случае листинг дли-

ной 2^{67} мм займет 15 световых лет, выходя далеко за пределы солнечной системы, до Альфы Центавра и обратно. Для отслеживания местонахождения процесса в каждом потоке используются 64-разрядные файловые указатели. Максимальный же размер потока составляет около 18,4 экзабайт[1].

Вызовы функций Win32 API для управления файлами и каталогами в первом приближении подобны соответствующим им двойникам в UNIX, но у функций Win32 API больше параметров и другая модель безопасности. Процедура открытия файла возвращает дескриптор файла, который затем может использоваться для чтения этого файла или записи в файл. Для графических приложений заранее не определены указатели в файлах. Стандартные потоки ввода, вывода и сообщений об ошибках при необходимости должны открываться явно. Однако в консольном режиме они открываются заранее. Интерфейс Win32 также содержит ряд дополнительных вызовов, отсутствующих в системе UNIX.

Вызовы API файловой системы в Windows 2000

Основные функции интерфейса Win32 API для работы с файлами перечислены в табл. 11.15. (Во второй колонке указывается ближайший эквивалент в системе UNIX.) На самом деле их значительно больше, но приведенные в таблице данные дают достаточное представление об основных функциях. Рассмотрим их кратко. С помощью функции `CreateFile` можно создать файл и получить дескриптор к нему. Эту же функцию следует применять и для открытия уже существующего файла, так как в Win32 API нет специальной функции `FileOpen`. В таблице не приводятся параметры функций, потому что они слишком многочисленны. Например, функция `CreateFile` имеет семь параметров, которые можно приближенно описать следующим образом:

1. Указатель на имя файла, который нужно создать или открыть.
2. Флаги (биты), указывающие, может ли с этим файлом выполняться чтение, запись или и то и другое.
3. Флаги, указывающие, может ли этот файл одновременно открываться несколькими процессами.
4. Указатель на описатель защиты, сообщающий, кто может получать доступ к файлу.
5. Флаги, сообщающие, что делать, если файл существует или, наоборот, не существует.
6. Флаги, управляющие архивацией, сжатием и т. д.
7. Дескриптор файла, чьи атрибуты должны быть клонированы для нового файла.

Следующие шесть функций в табл. 11.15 довольно близко совпадают с соответствующими им системными вызовами UNIX. Последние две функции позволяют заблокировать и разблокировать часть файла, чтобы гарантировать для процесса исключительный доступ к этому участку файла.

---

[1] 2^{64} байт составит 16 Эбайт ровно или 18 446 744 073 709 551 616 байт. Для байтов К означает не 1000, а 1024, М – не 1 000 000, а 1 048 576 и т. д. – *Примеч. перев.*

Таблица 11.15. Основные функции Win32 API для файлового ввода-вывода

Функция	UNIX	Описание
CreateFile	open	Создать или открыть файл; вернуть дескриптор файла
DeleteFile	unlink	Удалить существующий файл
CloseHandle	close	Закрыть файл
ReadFile	read	Прочитать данные из файла
WriteFile	write	Записать данные в файл
SetFilePointer	lseek	Установить указатель в файле в определенную позицию
GetFileAttributes	stat	Вернуть атрибуты файла
LockFile	fcntl	Заблокировать область файла для обеспечения взаимного исключения
UnlockFile	fcntl	Отменить блокировку области файла

С помощью этих функций API можно написать процедуру копирования файла, аналогичную версии для системы UNIX в листинге 6.1. Фрагмент этой программы (без какой бы то ни было проверки ошибок) приведен в листинге 11.1. Она была написана, чтобы сымитировать версию для системы UNIX. На практике копировать эту программу не нужно, так как в API существует функция CopyFile, выполняющая приблизительно те же действия.

Листинг 11.1. Фрагмент программы для копирования файла, использующей функции API Windows 2000

```
/* Открыть файлы для ввода и вывода. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL, NULL);
/* Копировать файл. */
do {
      s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
      if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);
/* Закрыть файлы. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Применяемая в операционной системе Windows 2000 файловая система NTFS представляет собой иерархическую файловую систему, сходную с файловой системой UNIX. Однако в качестве разделителя между именами компонентов пути вместо символа / используется символ \, атавизм, унаследованный от системы MS-DOS. В системе NTFS также существует понятие рабочего каталога, а пути могут быть относительными и абсолютными. Поддерживаются жесткие и символьные связи. Жесткие связи реализуются, как и в системе UNIX, при помощи нескольких записей в каталогах. Реализация символьных связей основана на точках повторного анализа (они будут обсуждаться ниже в этой главе). Поддерживаются сжатие, шифрование и устойчивость к сбоям. Эти свойства и их реализация также будут рассмотрены ниже.

Основные функции API для управления каталогами приведены в табл. 11.16. Как и в предыдущей таблице, во втором столбце приведены аналоги из системы UNIX.

Таблица 11.16. Основные функции Win32 API для управления каталогами

Функция	UNIX	Описание
CreateDirectory	mkdir	Создать новый каталог
RemoveDirectory	rmdir	Удалить пустой каталог
FindFirstFile	opendir	Инициализация, чтобы начать чтение записей каталога
FindNextFile	readdir	Прочитать следующую запись каталога
MoveFile	rename	Переместить файл из одного каталога в другой
SetCurrentDirectory	chdir	Изменить текущий рабочий каталог

Реализация файловой системы Windows 2000

Система NTFS представляет собой очень сложную файловую систему. Эта файловая система была полностью разработана заново, и она не является попыткой улучшить старую файловую систему MS-DOS. Ниже будет рассмотрено несколько ее особенностей, такие как структура, поиск файла по имени, сжатие и шифрование файлов.

Структура файловой системы

Каждый том NTFS (то есть дисковый раздел) содержит файлы, каталоги, битовые массивы и другие структуры данных. Каждый том организован как линейная последовательность блоков (кластеров по терминологии Microsoft). Размер блока фиксирован для каждого тома и варьируется в пределах от 512 байт до 64 Кбайт, в зависимости от размера тома. Для большинства дисков NTFS используются блоки размером в 4 Кбайт, как компромисс между большими блоками (для эффективности операций чтения/записи) и маленькими блоками (для уменьшения потерь дискового пространства на внутреннюю фрагментацию). Обращение к блокам осуществляется по их смещению от начала тома, для которого используются 64-разрядные числа.

Главной структурой данных в каждом томе является **главная файловая таблица MFT** (Master File Table), представляющая собой линейную последовательность записей фиксированного (1 Кбайт) размера. Каждая запись MFT описывает один файл или один каталог. В ней содержатся атрибуты файла, такие как его имя и временные штампы, а также список дисковых адресов, указывающих на расположение блоков файла. Если файл очень большой, то иногда бывает необходимо использовать две и более записей главной файловой таблицы, чтобы вместить список всех блоков файла. В этом случае первая запись MFT, называемая **базовой записью**, указывает на другие записи MFT. Такая избыточная схема берет свое начало в системе CP/M, в которой каждая каталоговая запись называлась экстентом. Какие из элементов главной файловой таблицы свободны, учитывается в битовом массиве.

Сама главная файловая таблица представляет собой файл и, как и любой файл, может располагаться в любом месте тома, тем самым устраняется проблема дефектных секторов на первой дорожке дискового раздела. Кроме того, этот файл может при необходимости расти до максимального размера в 2^{48} записей.

Структура главной файловой таблицы показана на рис. 11.17. Каждая запись MFT состоит из последовательности пар (заголовок атрибута, значение). Каждый атрибут начинается с заголовка, идентифицирующего этот атрибут и сообщающего длину значения, так как некоторые атрибуты, например имя файла или данные, могут иметь переменную длину. Если значение атрибута достаточно короткое, чтобы поместиться в запись MFT, оно помещается туда. Если же это значение слишком длинное, оно располагается в другом месте диска, а в запись MFT помещается указатель на него.

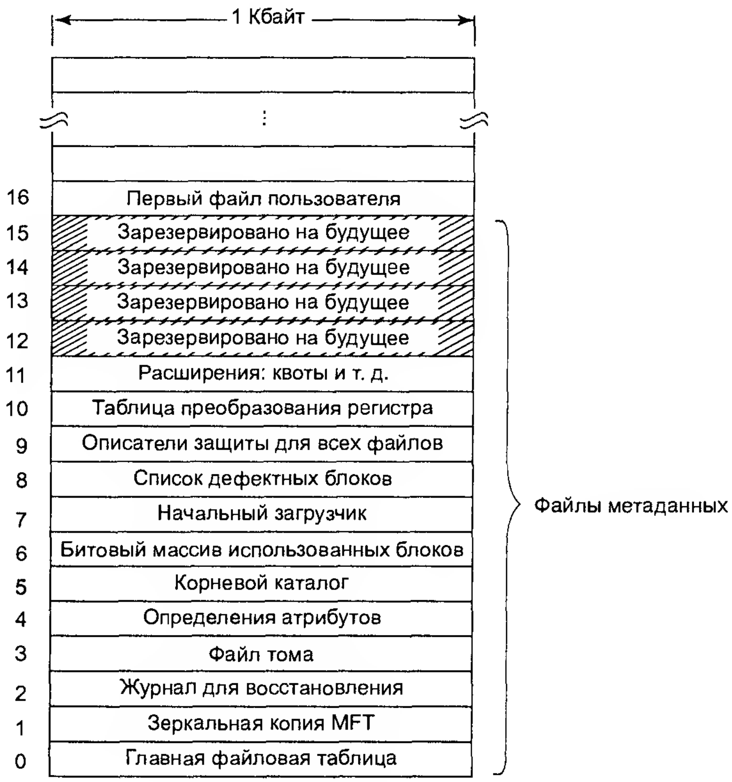


Рис. 11.17. Главная файловая таблица NTFS

Первые 16 записей MFT зарезервированы для файлов метаданных NTFS, как показано на рис. 11.17. Каждая запись описывает нормальный файл, у которого есть атрибуты и блоки данных, как у любого файла. У каждого такого файла есть имя, начинающееся с символа доллара, указывающего на то, что это файл метаданных. Первая запись описывает сам файл MFT. В частности, она содержит информацию о расположении блоков файла MFT, что позволяет системе найти файл MFT. Очевидно, чтобы найти всю остальную информацию о файловой системе, у операционной системы Windows 2000 должен быть некий способ нахождения первого блока файла MFT. Номер первого блока файла MFT содержится в загрузочном блоке, куда он помещается при установке системы.

Запись *1* представляет собой дубликат первой части файла MFT. Эта информация является настолько ценной, что наличие второй копии может быть необходимо на случай, если один из первых блоков главной файловой таблицы вдруг станет дефектным. Запись *2* представляет собой журнал. Когда в файловой системе производятся структурные изменения, такие как добавление нового каталога или удаление существующего каталога, информация о предстоящей операции регистрируется в журнале. Таким образом, увеличивается вероятность корректного восстановления файловой системы в случае сбоя во время выполнения операции. Изменения атрибутов файлов также регистрируются здесь. В этом журнале не регистрируются только изменения данных пользователя. В записи *3* содержится информация о томе, например его размер, метка и версия.

Как уже сообщалось, каждая запись MFT содержит последовательность пар (заголовок атрибута, значение). Файл *$AttrDef* является тем местом, в котором определяются атрибуты. Информация об этом файле хранится в записи *4* таблицы MFT. В следующей записи содержатся данные о корневом каталоге, который сам представляет собой файл и может произвольно увеличиваться в размерах. Он описывается записью *5* главной файловой таблицы.

Свободное место на диске учитывается с помощью битового массива. Битовый массив сам является файлом, и его атрибуты и дисковые адреса хранятся в записи *6* таблицы MFT. Следующая запись таблицы MFT указывает на файл начальной загрузки. Запись *8* используется для того, чтобы связать вместе все дефектные блоки и гарантировать, что они никогда не встретятся в файлах. Запись *9* содержит информацию о защите. Запись *10* используется для преобразования регистра. Для символов латинского алфавита от A до Z преобразование регистра не представляет проблем (по крайней мере для тех, кто говорит на языке с латиницей). Для других языков, таких как греческий, армянский или грузинский, этот вопрос не столь очевиден, как для использующих латынь, поэтому этот файл содержит необходимые инструкции. Наконец, запись *11* представляет собой каталог, содержащий различные файлы для дисковых квот, идентификаторов объектов, точек повторного анализа и т. д. Последние четыре записи MFT зарезервированы на будущее.

Каждая запись MFT состоит из заголовка записи, за которым следует последовательность пар (заголовок атрибута, значение). Заголовок записи содержит магическое число, используемое для проверки действительности записи; порядковый номер, обновляемый каждый раз, когда запись используется для нового файла; счетчик обращений к файлу; действительное количество байт, используемых в записи; идентификатор (индекс, порядковый номер) базовой записи (используемый только для записей расширения), а также другие различные поля. Следом за заголовком записи располагается заголовок первого атрибута, за которым идет значение первого атрибута, потом заголовок второго атрибута, значение второго атрибута и т. д.

В файловой системе NTFS определено 13 атрибутов, которые могут появляться в записях MFT. Они перечислены в табл. 11.17. Все записи таблицы MFT состоят из последовательности заголовков атрибутов, каждый из которых идентифицирует следующий за ним атрибут, а также содержит длину и расположение поля значения вместе с разнообразными флагами и прочей информацией.

Как правило, значения атрибутов располагаются непосредственно за заголовками, но если длина значения слишком велика, чтобы поместиться в запись таблицы MFT, она может быть помещена в отдельный блок диска. Такой атрибут называется **нерезидентным атрибутом**. Например, таким атрибутом является атрибут данных. Некоторые атрибуты, такие как атрибуты имени, могут повторяться, но все атрибуты должны располагаться в записи MFT в фиксированном порядке. Длина заголовков резидентных атрибутов 24 байт, заголовки для нерезидентных атрибутов длиннее, так как они содержат информацию о месте расположения атрибута.

Таблица 11.17. Атрибуты, используемые в записях MFT

Атрибут	Описание
Стандартная информация	Флаговые биты, временные штампы и т. д.
Имя файла	Имя файла в кодировке Unicode; может быть также повторено для имени MS-DOS
Описатель защиты	Устарел. Теперь информация о защите располагается в атрибуте $Extend$Secure
Список атрибутов	Расположение дополнительных записей MFT
Идентификатор объекта	64-разрядный идентификатор файла, уникальный для данного тома
Точка повторного анализа	Используется для монтирования и символьных ссылок
Название тома	Название тома (используется только в $Volume)
Информация о томе	Версия тома (используется только в $Volume)
Корневой индекс	Используется для каталогов
Размещение индекса	Используется для очень больших каталогов
Битовый массив	Используется для очень больших каталогов
Поток данных утилиты регистрации	Управляет регистрацией в файле $LogFile
Данные	Поточные данные; может повторяться

Стандартное информационное поле содержит сведения о владельце файла, информацию о защите, временные штампы, необходимые для стандарта POSIX, счетчик жестких связей, бит «только чтение», архивный бит и т. д. Это поле имеет фиксированную длину, и оно всегда присутствует. Имя файла хранится в кодировке Unicode в поле переменной длины. Чтобы старые 16-разрядные программы могли работать с файлами, файлы также могут содержать имена формата MS-DOS 8+3. Если имя файла удовлетворяет правилам именования файлов в MS-DOS, второе имя формата MS-DOS не используется.

В операционной системе NT 4.0 информация о защите файла могла содержаться в атрибуте файла, но в Windows 2000 эти данные хранятся в отдельном файле, что позволяет нескольким файлам совместно пользоваться общими описателями защиты. Список атрибутов нужен на случай, если атрибуты не помещаются в запись MFT. Каждая запись в списке содержит 48-разрядный индекс в таблице MFT, указывающий на запись расширения, а также 16-разрядный порядковый номер, позволяющий проверить соответствие записи расширения и базовой записи.

Атрибут идентификатор объекта задает файлу уникальный номер. Иногда он используется внутри системы. Точка повторного анализа велит процедуре, анали-

зирующей имя файла, выполнить специальные действия. Этот механизм применяется для монтирования устройств и символьных ссылок. Два следующих атрибута используются только для идентификации тома. Еще три атрибута используются для реализации каталогов. Небольшие каталоги представляют собой простые списки файлов, но большие каталоги реализуются в виде деревьев B+. Поток данных утилиты регистрации используется шифрующей файловой системой.

Наконец, мы добрались до атрибута, которого все так ждали: до атрибута данных. Имя потока данных, если оно присутствует, располагается в этом заголовке атрибута. Следом за этим заголовком располагается либо список дисковых адресов, определяющий положение файла на диске, либо, для файлов длиной всего в несколько сотен байтов (а таких файлов довольно много), сам файл. Метод помещения самого содержимого файла в запись MFT называется **непосредственным файлом** [241].

Конечно, в большинстве случаев все данные файла не помещаются в запись MFT, поэтому этот атрибут, как правило, является нерезидентным. Рассмотрим теперь, как в файловой системе NTFS отслеживается расположение нерезидентных атрибутов, в частности данных.

Для увеличения эффективности дисковые блоки файлам назначаются по возможности в виде серий последовательных блоков (сегментов файла). Например, если первый логический блок файла помещается в блоке 20 на диске, тогда система будет стараться выделить для второго блока этого файла блок 21, для третьего — блок 22 и т. д. Один из способов выделения файлам таких серий блоков заключается в том, чтобы предоставлять файлам сразу по несколько блоков.

Блоки в файле описываются последовательностью записей, каждая из которых описывает последовательность логически непрерывных блоков. Непрерывный файл описывается всего одной записью. К этой категории относятся файлы, записываемые за одну операцию от начала до конца. Файл с одной «дыркой» (например файл, для которого определены только блоки с 0 по 49 и с 60 по 79), будет описываться двумя записями. Такой файл может быть создан, если сначала записать в него первые 50 блоков, затем переместить указатель в файле на логический блок 60 и записать еще 20 блоков. Когда из такого файла читается «дырка», все отсутствующие байты оказываются нулями.

Каждая запись начинается с заголовка, определяющего смещение первого блока в файле. Затем располагается смещение первого блока, не покрываемого первой записью. Для приведенного выше примера у первой записи будет заголовок (0, 50), а сама запись будет содержать дисковые адреса для первых 50 блоков файла. Вторая запись будет иметь заголовок (60, 80) и содержать дисковые адреса для следующих 20 блоков файла.

Следом за каждым заголовком располагаются пары, в которых содержатся дисковые адреса и длины серий блоков. Эти дисковые адреса представляют собой смещение блока от начала дискового раздела. Длина серии — это количество блоков в серии. В записи серии может содержаться любое необходимое количество пар. Использование этой схемы для 9-блочного файла, состоящего из трех сегментов, проиллюстрировано на рис. 11.18.

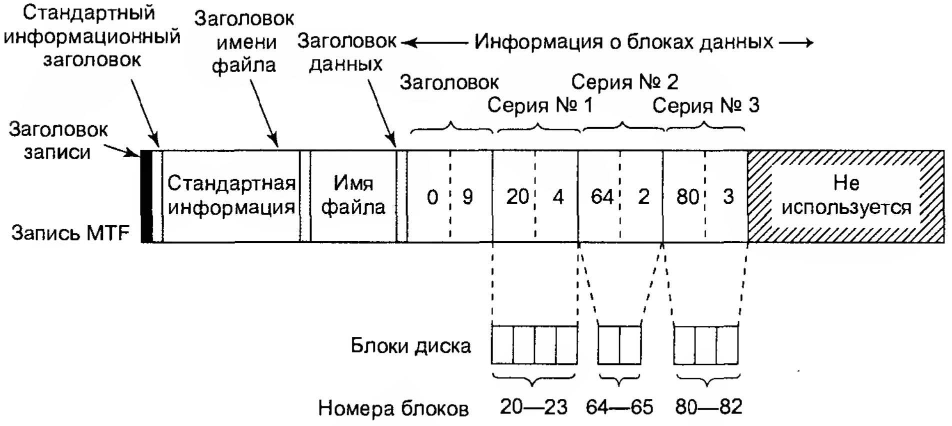


Рис. 11.18. Запись MFT для 9-блочного файла, состоящего из трех сегментов

На этом рисунке показана запись MFT для короткого файла («короткий файл» здесь означает, что вся информация о блоках файла помещается в одну запись MFT). Файл состоит из трех серий последовательных блоков диска. Первая серия блоков располагается в блоках диска с 20 по 23, вторая — в 64 и 65, и третья — с 80 по 82. Каждая серия записывается в записи MFT в виде пары (дисковый адрес, количество блоков). Число таких серий зависит от того, насколько удачно процедура предоставления дискового пространства сумела найти место для хранения файла при его создании. Для файла, состоящего из n блоков, количество серий может быть любым от 1 до n.

Здесь следует сказать несколько слов. Во-первых, данный способ представления информации о расположении блоков файла на диске не накладывает никаких дополнительных ограничений на размер файла. При отсутствии сжатия адреса для каждой пары требуется два 64-разрядных числа, что составляет 16 байт на пару. Однако одна пара может указывать на миллион последовательных блоков. Например, 20-мегабайтный файл, состоящий из 20 сегментов по 1 млн килобайтных блоков каждый, легко может быть описан всего одной записью MFT, тогда как 60-килобайтный файл, состоящий из 60 изолированных блоков, не может.

Во-вторых, хотя при использовании простого метода представления пары требуется 2×8 байт, эти 16 байт могут быть сжаты до меньшего размера. Многие дисковые адреса содержат большое количество нулей в старших байтах. Нули могут быть опущены. В этом случае в заголовке данных будет содержаться информация о том, сколько байтов пропущено, то есть сколько байтов фактически используется для дискового адреса. Также используются и другие методы сжатия данных. На практике пары часто занимают всего 4 байт.

Наш первый пример был довольно прост: вся информация о файле помещалась в одну запись MFT. Что произойдет, если файл окажется настолько велик или будет настолько фрагментирован, что информация о блоках не поместится в одну запись MFT? Ответ прост: нужно использовать две или более записей MFT. На рис. 11.19 показан файл, базовая запись которого располагается в записи 102 таблицы MFT. Этот файл состоит из слишком большого количества сегментов, чтобы информация о них могла поместиться в одну запись MFT, поэтому для

этого файла вычисляется необходимое ему число записей расширения — например, две, — и их индексы помещаются в базовую запись. Оставшаяся часть записи используется для адресов первых *k* сегментов файла.

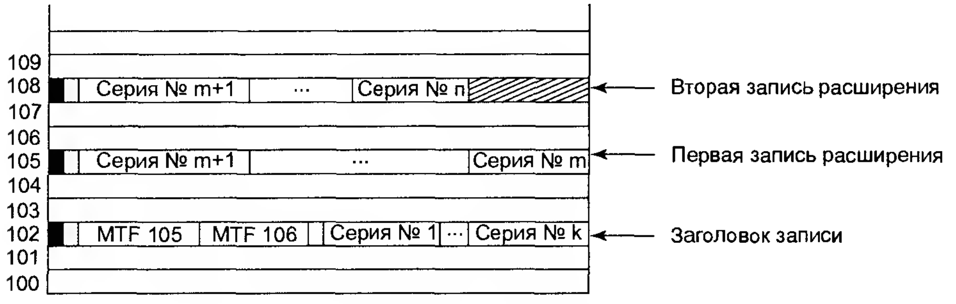


Рис. 11.19. Файл, которому требуется три записи MFT для хранения данных обо всех своих сегментах

Обратите внимание, что схема на рис. 11.19 содержит некоторую избыточную информацию. Теоретически необходимости указывать конец последовательности сегментов нет, так как эта информация может быть вычислена по сегментным парам. Причина, по которой эта информация дублируется, заключается в повышении эффективности поиска блока файла. Чтобы найти блок по смещению в файле, нужно всего лишь изучить заголовки записей, а не сами сегментные пары.

Когда все пространство в записи 102 использовано, сегментные пары помещаются в запись 105. В эту запись помещается столько пар, сколько туда влезет. Когда и эта запись оказывается заполненной, остальные пары помещаются в запись 108. Таким образом, для хранения больших фрагментированных файлов может быть использовано несколько записей таблицы MFT.

Проблема может возникнуть, если потребуется так много записей MFT, что в базовой записи не поместятся все индексы MFT. Эта проблема решается следующим образом: список записей MFT делается нерезидентным (то есть хранится отдельно на диске, а не в базовой записи MFT). В этом случае его размер уже ничем не ограничен.

Запись MFT для небольшого каталога показана на рис. 11.20. Запись MFT содержит несколько каталоговых записей, каждая из которых описывает файл или каталог. Фиксированная часть содержит индекс записи MFT файла, длину имени файла, а также другие разнообразные поля и флаги. Поиск файла в каталоге по имени состоит в последовательном переборе всех имен файлов.

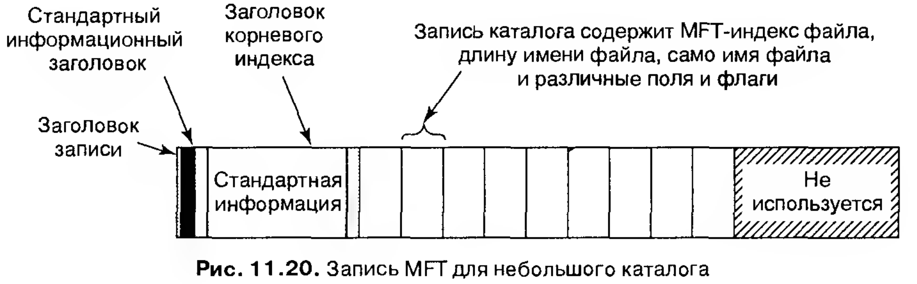


Рис. 11.20. Запись MFT для небольшого каталога

Для больших каталогов используется другой формат. Вместо того чтобы линейно перечислять файлы, используется дерево B+, обеспечивающее поиск в алфавитном порядке и упрощающее добавление в каталог новых имен в соответствующие места.

Поиск файла по имени

Теперь мы обладаем достаточной информацией, чтобы понять, как осуществляется поиск файла по имени. Когда пользовательская программа пытается открыть файл, она обычно обращается к библиотечной процедуре вроде следующей:

```
CreateFile("C:\maria\web.htm", ...)
```

Этот вызов попадает в совместно используемую библиотеку уровня пользователя *kernel32.dll*, где *\??* помещается перед именем файла, в результате чего получается строка

```
\??\C:\maria\web.htm
```

Это имя пути передается системному вызову `NtFileCreate` в качестве параметра.

Затем операционная система начинает поиск в корне пространства имен менеджера объектов (см. табл. 11.7). После этого она ищет и находит *C:* в каталоге *\??*. Этот файл представляет собой символьную ссылку на другую часть пространства имен менеджера объектов, на каталог *\Device*. Ссылка, как правило, указывает на объект с именем вроде *\Device\HarddiskVolume1*. Этот объект соответствует первому разделу первого жесткого диска. По объекту можно определить, какую таблицу MFT, располагающуюся на этом дисковом разделе, следует использовать. Этапы поиска файла показаны на рис. 11.21.

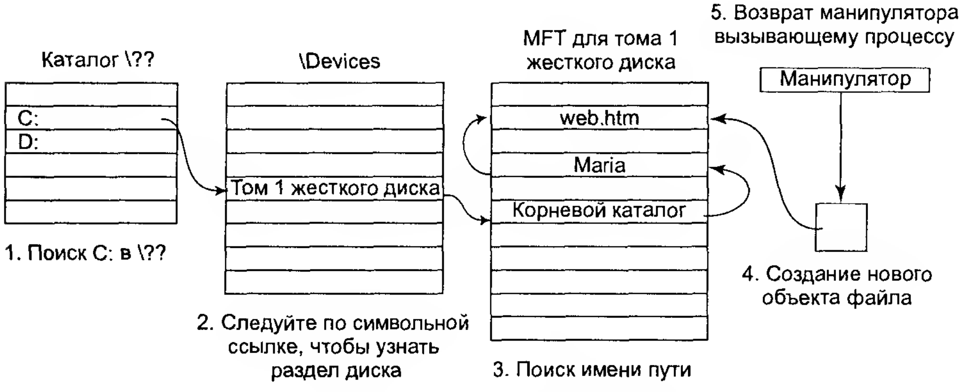


Рис. 11.21. Этапы поиска файла C:\maria\web.htm

Анализ имени файла продолжается теперь в корневом каталоге дискового раздела, блоки которого можно найти в записи *5* таблицы MFT (см. рис. 11.17). Затем в корневом каталоге ищется строка «maria», в результате чего процесс получает индекс в таблице MFT для каталога *maria*. Затем в каталоге maria ищется строка «web.htm». Если поиск завершается успешно, то результатом его является новый объект, созданный менеджером объектов. Этот объект, не имеющий имени, содер-

жит индекс в таблице MFT для найденного файла. Вызывающему процессу возвращается дескриптор этого объекта. При последующих вызовах ReadFile в качестве входного параметра используется дескриптор объекта, что позволяет менеджеру объектов найти индекс, а затем и содержимое записи MFT для этого файла. Если этот же файл откроет второй поток, ему будет предоставлен дескриптор нового объекта файла.

Помимо обычных файлов и каталогов, файловая система NTFS поддерживает жесткие связи, подобные используемым в UNIX, а также символьные ссылки при помощи механизма, называемого **точками повторного анализа**. Файл или каталог может быть помечен как точка повторного анализа, и с ним может быть ассоциирован блок данных. Когда во время анализа имени файла встречается такой файл или каталог, срабатывает обработка исключения и интерпретируется блок данных. Блок может выполнять различные действия, включая переадресацию поиска, ссылаясь на другую часть дерева каталогов или даже на другой дисковый раздел. Этот механизм используется для поддержки как символьных ссылок, так и монтировки файловых систем.

Сжатие файлов

Файловая система NTFS поддерживает прозрачное сжатие файлов. Файл может быть создан в сжатом режиме. Это означает, что файловая система NTFS будет автоматически пытаться сжать блоки этого файла при записи их на диск и автоматически распаковывать их при чтении. Процессы, читающие этот файл или пишущие в него, не будут даже догадываться о том, что при этом происходит компрессия или декомпрессия данных.

Сжатие данных файла происходит следующим образом. Когда файловая система NTFS записывает на диск файл, помеченный для сжатия, она изучает первые 16 (логических) блоков файла, независимо от того, сколько сегментов на диске они занимают. Затем к этим блокам применяется алгоритм сжатия. Если полученные на выходе блоки могут поместиться в 15 или менее блоков, то сжатые данные записываются на диск, предпочтительно в виде одного сегмента. Если получить выигрыш хотя бы в один блок не удается, то данные 16 блоков так и записываются в несжатом виде. Затем весь алгоритм повторяется для следующих 16 блоков и т. д.

На рис. 11.22, *а* показан файл, в котором первые 16 блоков успешно сжаты в 8 блоков, следующие 16 блоков не могут быть сжаты, наконец, последние 16 блоков также успешно сжаты на 50 %. Эти три части файла записаны в виде трех сегментов, информация о которых хранится в записи MFT. «Пропущенные» блоки обозначаются в записи MFT как сегменты с нулевым дисковым адресом (рис. 11.22, *б*). Здесь за заголовком (0,48) следует пять пар, две для первого (сжатого) сегмента, одна для несжатого сегмента и две для последнего (сжатого) сегмента.

При чтении этого файла система NTFS должна знать, какие из сегментов файла сжаты, а какие нет. Она видит это по дисковым адресам. Дисковый адрес 0 указывает на то, что предыдущий сегмент сжат. Дисковый блок 0 не может использоваться для хранения данных во избежание неоднозначности. Поскольку в этом блоке содержится загрузочный сектор, использование этого блока для хранения данных все равно невозможно.

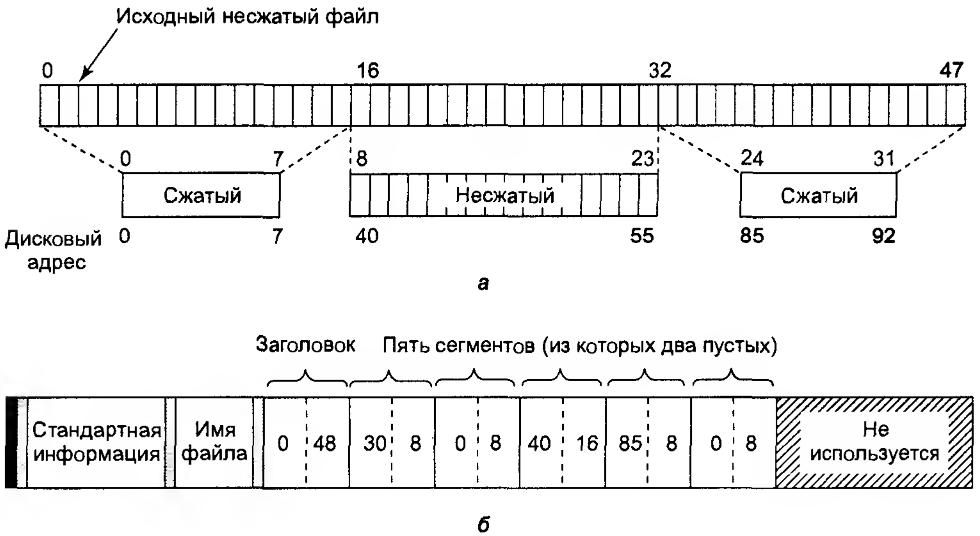


Рис. 11.22. Пример 48-блочного файла, сжатого до 32 блоков (*а*); запись MFT после сжатия файла (*б*)

Произвольный доступ к сжатому файлу возможен, но не совсем прост. Предположим, процесс обращается к блоку 35 файла, показанного на рис. 11.22. Как файловой системе NTFS найти блок 35 в сжатом файле? Ответ состоит в том, что для этого сначала потребуется прочитать и распаковать весь сегмент файла. После этого система может определить, где находится блок 35, и передать его читающему процессу. Сжатие файла частями именно по 16 блоков явилось компромиссом. Если бы файл сжимался меньшими порциями, эффективность сжатия снизилась бы. Выбор больших размеров сжимаемых фрагментов привел бы к замедлению произвольного доступа к блокам.

Шифрование файлов

Сегодня компьютеры используются для хранения самых разнообразных конфиденциальных данных, включая планы слияния корпораций, налоговую информацию и любовную переписку. Владельцы подобных данных, как правило, не желают, чтобы она попала в посторонние руки. Информация может оказаться потеряна, например, при потере или краже переносного компьютера. Настольный компьютер можно загрузить с гибкого диска с системой MS-DOS, чтобы обойти систему безопасности Windows 2000. Наконец, жесткий диск можно просто вынуть из одного компьютера и установить на другой компьютер. Очень опасной может оказаться даже прогулка в туалет, если компьютер будет оставлен без присмотра во включенном состоянии.

В операционной системе Windows 2000 эти проблемы решаются при помощи возможности шифрования файлов. В результате применения шифрования, даже если компьютер будет украден или перезагружен в системе MS-DOS, файлы останутся нечитаемыми. Чтобы использовать шифрование в операционной системе Windows 2000, нужно пометить каталог как зашифрованный, в результате чего

будут зашифрованы все файлы в этом каталоге, а все новые файлы, перемещенные в этот каталог или созданные в нем, также будут зашифрованы. Само шифрование и дешифрование выполняется не файловой системой NTFS, а специальным драйвером **EFS** (Encrypting File System — шифрующая файловая система), размещающимся между NTFS и пользовательским процессом. Таким образом, прикладная программа не знает о шифровании, а сама файловая система NTFS только частично вовлечена в этот процесс.

Чтобы понять, как работает система шифрования файлов, необходимо понимать, как работает современная система шифрования. Для этого в разделе «Основы криптографии» главы 9 был дан краткий обзор данной темы. Читатели, не знакомые с основами криптографии, должны сначала прочитать этот раздел.

Познакомимся теперь с тем, как шифруются файлы в операционной системе Windows 2000. Когда пользователь сообщает системе, что хочет зашифровать определенный файл, формируется случайный 128-разрядный ключ. Ключ используется для поблочного шифрования файла с помощью симметричного алгоритма, параметром в котором используется этот ключ. Каждый новый шифруемый файл получает новый случайный 128-разрядный ключ, так что никакие два файла не используют один и тот же ключ шифрования, что увеличивает защиту данных в случае, если какой-либо из ключей окажется скомпрометированным. Применяемый в настоящий момент алгоритм шифрования представляет собой вариант стандартного алгоритма **DES** (Data Encryption Standard — стандарт шифрования данных), но архитектура EFS поддерживает добавление новых алгоритмов в будущем. Независимое шифрование каждого блока файла необходимо для сохранения возможности произвольного доступа к блокам файла.

Чтобы файл мог быть впоследствии расшифрован, ключ файла должен где-то храниться. Если бы ключ хранился на диске в открытом виде, тогда злоумышленник, укравший файлы, мог бы легко найти его и воспользоваться им для расшифровки украденных файлов. В этом случае сама идея шифрования файлов оказалась бы бессмысленной. Поэтому ключи файлов сами должны храниться на диске в зашифрованном виде. Для этого используется шифрование с открытым ключом.

После того как файл зашифрован, система с помощью информации в системном реестре ищет расположение открытого ключа пользователя. Открытый ключ можно без каких-либо опасений хранить прямо в реестре, так как по открытому ключу невозможно определить закрытый ключ, необходимый для расшифровки файлов. Затем случайный 128-разрядный ключ файла шифруется открытым ключом, а результат сохраняется на диске вместе с файлом, как показано на рис. 11.23.

Чтобы расшифровать файл, с диска считывается зашифрованный случайный 128-разрядный ключ файла. Однако для его расшифровки необходим закрытый ключ. В идеале этот ключ должен храниться на смарт-карте, вне компьютера, и вставляться в считывающее устройство только тогда, когда требуется расшифровать файл. Хотя операционная система Windows 2000 поддерживает смарт-карты, она не позволяет хранить на них закрытые ключи.

Вместо этого, когда пользователь в первый раз зашифровывает файл с помощью системы EFS, операционная система Windows 2000 формирует пару ключей (закрытый ключ, открытый ключ) и сохраняет закрытый ключ, зашифрованный при помощи симметричного алгоритма шифрования, на диске. Ключ для этого

симметричного алгоритма формируется либо из пароля пользователя для регистрации в системе, либо из ключа, хранящегося на смарт-карте, если регистрация при помощи смарт-карты разрешена. Таким образом, система EFS может расшифровать закрытый ключ во время регистрации пользователя в системе и хранить его в своем виртуальном адресном пространстве во время работы, чтобы иметь возможность расшифровывать 128-разрядные ключи файлов без дополнительного обращения к диску. Когда компьютер выключается, закрытый ключ стирается из виртуального адресного пространства системы EFS, так что никто, даже украв компьютер, не получит доступа к закрытому ключу.

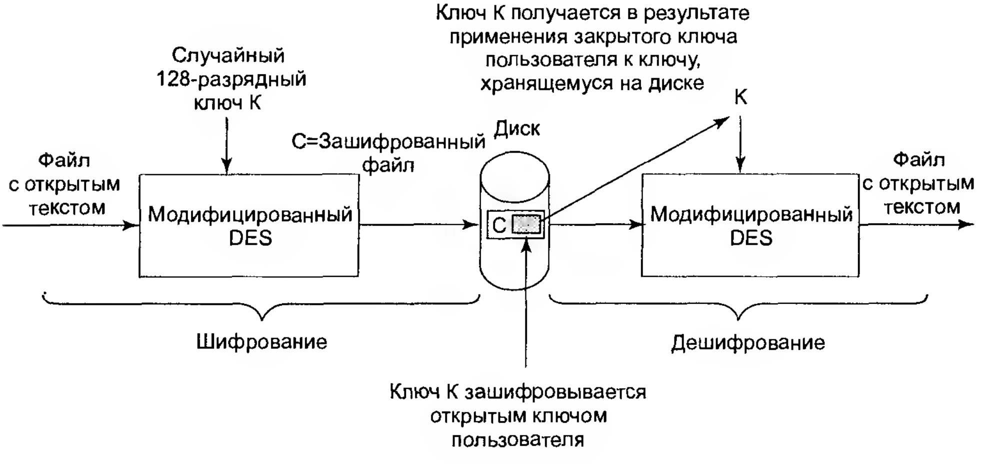


Рис. 11.23. Работа системы шифрования файлов

Сложности возникают тогда, когда нескольким пользователям требуется доступ к одному и тому же зашифрованному файлу. В настоящий момент совместное использование зашифрованных файлов несколькими пользователями не поддерживается. Однако в будущем архитектура EFS может поддержать совместное использование, если ключ шифрования файла будет зашифровываться несколько раз, отдельно для каждого авторизованного пользователя, его закрытым ключом. Все зашифрованные версии ключа могут добавляться к файлу.

Потенциальная потребность в совместном использовании зашифрованных файлов является одной из причин, по которой применяется такая двухуровневая система ключей. Если бы все файлы зашифровывались ключами владельцев файлов, то совместное использование зашифрованных файлов было бы невозможным. Эта проблема может быть решена, если для зашифровки каждого файла использовать отдельный ключ.

Схема с использованием случайных ключей для шифрования файлов, но с шифрованием самих ключей при помощи симметричного алгоритма шифрования не будет работать. Проблема в том, что наличие симметричного ключа, хранящегося на диске в открытом виде, разрушит всю систему защиты — сформировать ключ дешифрации по ключу шифрования слишком легко. Таким образом, медленное шифрование с открытым ключом требуется для шифрования ключей файлов.

Поскольку ключи шифрования все равно являются открытыми, хранение их в открытом виде не представляет опасности.

Вторая причина использования двухуровневой системы ключей заключается в производительности. Использование криптографии с открытым ключом для шифрования файлов было бы слишком медленным. Для повышения эффективности шифрование с открытым ключом применяется лишь для зашифровки коротких 128-разрядных ключей файлов, тогда как для шифрования самих файлов используется симметричный алгоритм.

Безопасность в Windows 2000

Познакомившись с шифрованием в файловой системе, рассмотрим теперь систему безопасности в Windows 2000 в целом. Операционная система NT была разработана так, чтобы соответствовать уровню С2 требований безопасности Министерства обороны США (DoD 5200.28-STD), Оранжевой книги, обсуждавшейся в главе 9. Этот стандарт требует наличия у операционных систем определенных свойств, позволяющих относить данные системы к достаточно надежным для выполнения военных задач определенного рода. Хотя при разработке операционной системы Windows 2000 не ставилось особой цели соответствия требованиям уровня С2, она унаследовала множество свойств безопасности от NT, включая следующие:

1. Безопасная регистрация в системе с мерами предосторожности против попыток применения фальшивой программы регистрации.
2. Дискреционное управление доступом.
3. Управление привилегированным доступом.
4. Защита адресного пространства для каждого процесса.
5. Обнуление страниц перед выделением их процессу.
6. Аудит безопасности.

Рассмотрим кратко эти аспекты (ни один из них не встречается в Windows 98).

Безопасная регистрация означает, что системный администратор может потребовать ото всех пользователей наличия пароля для входа в систему. Программа, имитирующая регистрацию в системе, использовалась ранее на некоторых системах злоумышленниками с целью выведать пароль пользователя. Такая программа запускалась в надежде, что пользователь сядет за компьютер и введет свое имя и пароль. Имя и пароль записывались на диск, после чего пользователю сообщалось, что в регистрации ему отказано. В операционной системе Windows 2000 подобный обман пользователя невозможен, так как пользователь для входа в систему должен нажать комбинацию клавиш CTRL+ALT+DEL. Эта комбинация клавиш всегда перехватывается драйвером клавиатуры, который вызывает при этом настоящую программу регистрации. Пользовательский процесс не может сам перехватить эту комбинацию клавиш или отменить ее обработку драйвером.

Дискреционное управление доступом позволяет владельцу файла или другого объекта указать, кто может пользоваться объектом и каким образом. Средства

управления привилегированным доступом позволяют системному администратору (суперпользователю) получать доступ к объекту, несмотря на установленные его владельцем разрешения доступа. Под защитой адресного пространства имеется в виду лишь то, что у каждого процесса есть собственное защищенное виртуальное адресное пространство, недоступное для любого неавторизованного процесса. Следующий пункт означает, что при увеличении стека выделяемые для него страницы заранее обнуляются, так что процесс не может обнаружить в них информации, помещенной предыдущим владельцем страницы памяти (страницы подаются процессам из списка обнуленных страниц, показанного на рис. 11.14). Наконец, аудит безопасности следует понимать как регистрацию системой в журнале определенных событий, относящихся к безопасности. Впоследствии этот журнал может просматривать системный администратор.

В следующем разделе будут описаны основные понятия безопасности операционной системы Windows 2000. После этого мы обсудим системные вызовы безопасности. Наконец, мы рассмотрим вопросы реализации системы безопасности в Windows 2000.

Основные понятия

У каждого пользователя (и группы) операционной системы Windows 2000 есть идентификатор безопасности **SID** (Security IDentifier), по которому операционная система отличает его от других пользователей. Идентификаторы безопасности представляют собой двоичные числа с коротким заголовком, за которым следует длинный случайный компонент. Каждый SID должен быть уникален в пределах всей планеты. Когда пользователь запускает процесс, этот процесс и его потоки работают под идентификатором пользователя. Большая часть системы безопасности спроектирована так, чтобы гарантировать предоставление доступа к каждому объекту только потокам с авторизованными идентификаторами безопасности.

У каждого процесса есть **маркер доступа**, в котором указывается SID и другие свойства. Как правило, он назначается при регистрации в системе процедурой *winlogon*. Структура маркера доступа показана на рис. 11.24. Чтобы получить эту информацию, процесс должен вызвать функцию *GetTokenInformation*, так как она может измениться со временем. Заголовок маркера содержит некоторую административную информацию. По значению поля срока действия можно определить, когда маркер перестанет быть действительным, но в настоящее время это поле не используется. Поле *Groups* (группы) указывает группы, к которым принадлежит процесс. Это поле необходимо для соответствия требованиям стандарта POSIX. Поле *Default DACL* (**DACL** по умолчанию, Discretionary Access Control List — список разграничительного контроля доступа) представляет собой список управления доступом, назначаемый объектам, созданным процессом, если не определены другие списки ACL. Идентификатор безопасности пользователя указывает пользователя, владеющего процессом. Ограниченные идентификаторы SID позволяют ненадежным процессам принимать участие в заданиях вместе с надежными процессами, но с меньшими полномочиями и меньшими возможностями причинения ущерба.

Заголовок	Срок действия	Группы	DACL	Ограниченные идентификаторы SID	SID пользователя	SID группы	Привилегии

Рис. 11.24. Структура маркера доступа

Наконец, перечисленные в маркере привилегии (если они перечислены) дают процессу особые полномочия, такие как право выключать компьютер или получать доступ к файлам, к которым в противном случае в этом доступе процессу было бы отказано. Привилегии позволяют разбить полномочия системного администратора на отдельные права, которые могут предоставляться процессам по отдельности. Таким образом, пользователю может быть предоставлена часть полномочий суперпользователя, но не все его полномочия. Итак, маркер доступа содержит информацию о том, кто владеет процессом и какие умолчания и полномочия ассоциированы с ним.

Когда пользователь регистрируется в системе, процесс *winlogon* назначает маркер доступа начальному процессу. Последующие процессы, как правило, наследуют этот маркер. Маркер доступа процесса изначально применяется ко всем потокам процесса. Однако поток во время исполнения может получить другой маркер доступа. В этом случае маркер доступа потока перекрывает маркер доступа процесса. В частности, клиентский поток может передать свой маркер доступа серверному потоку, чтобы сервер мог получить доступ к защищенным файлам и другим объектам клиента. Такой механизм называется **перевоплощением**.

Другим основным понятием является **дескриптор защиты**. У каждого объекта есть ассоциированный с ним дескриптор защиты, содержащий список пользователей и групп, имеющих доступ к данному объекту. Дескриптор защиты состоит из заголовка, за которым следует список DACL с одним или несколькими элементами **ACE** (Access Control Entry — элемент списка контроля доступа ACL). Два основных типа элементов списка — это разрешение и запрет доступа. Разрешающий элемент содержит SID пользователя или группы и битовый массив, определяющий набор операций, которые процессы с данным идентификатором SID могут выполнять с определенным объектом. Запрещающий элемент работает аналогично, но совпадение идентификаторов означает, что обращающийся процесс не может выполнять перечисленные операции. Например, у Иды есть файл, дескриптор защиты которого указывает, что у всех пользователей есть доступ для чтения этого файла, Элвису запрещен всякий доступ, а у самой Иды есть все виды доступа. Этот простой пример показан на рис. 11.25. Идентификатор защиты Everyone (все) соответствует множеству всех пользователей, но его действие может перекрываться любым элементом списка, в котором явно указано нечто иное.

Кроме списка DACL у дескриптора защиты есть также список **SACL** (System Access Control List — системный список контроля доступа), который похож на DACL, только вместо пользователей и групп, имеющих доступ к объекту, в нем перечисляются операции с этим объектом, регистрируемые в специальном журнале. На рис. 11.25 все действия, которые Мэрилин выполнит с этим файлом, будут регистрироваться в журнале. В операционной системе Windows 2000 также предоставляются дополнительные возможности аудита для регистрации доступа к объектам.

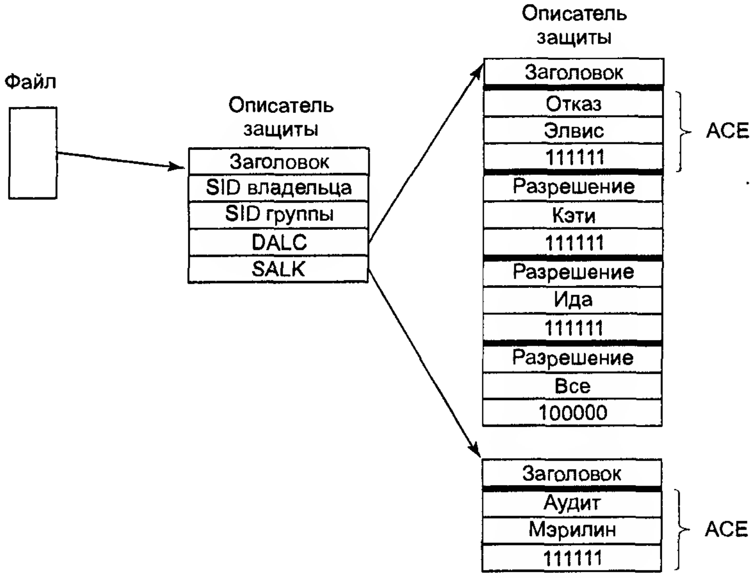


Рис. 11.25. Пример дескриптора защиты для файла

Вызовы API защиты

В основе большей части механизмов управления доступом в Windows 2000 лежат дескрипторы защиты. Как правило, когда процесс создает объект, он передает функции `CreateProcess`, `CreateFile` или другой функции в качестве одного из параметров дескриптор защиты. Этот дескриптор защиты затем становится дескриптором защиты, присоединенным к объекту (см. рис. 11.25). Если при создании объекта не предоставляется дескриптора защиты, используется дескриптор защиты вызывающего процесса по умолчанию (см. рис. 11.24).

Для управления дескрипторами защиты существует множество вызовов Win32 API. Наиболее важные вызовы перечислены в табл. 11.18. Чтобы создать дескриптор защиты, для него сначала выделяется место хранения, после чего он инициализируется с помощью вызова `InitializeSecurityDescriptor`. Этот вызов заполняет заголовок. Если SID владельца неизвестен, он может быть найден по имени при помощи вызова `LookupAccountSid`. Затем он может быть вставлен в дескриптор защиты. То же самое справедливо для SID группы, если группа существует. Как правило, используется SID вызывающего процесса, а также SID одной из групп вызывающего процесса, но системный администратор может записать в дескриптор защиты любой SID.

Список DACL или SACL дескриптора защиты может быть проинициализирован при помощи функции `InitializeAcl`. Элементы списка ACL могут быть добавлены с помощью функций `AddAccessAllowedAce` и `AddAccessDeniedAce`. К этим вызовам можно обращаться многократно, чтобы добавить столько записей ACE, сколько необходимо. Удаление записи происходит при помощи функции `DeleteAce`. Когда список ACL готов, его можно присоединить к дескриптору защиты с помощью

функции SetSecurityDescriptorDacl. Наконец, когда объект создан, к нему можно присоединить новенький дескриптор защиты.

Таблица 11.18. Основные функции Win32 API для управления защитой

Функция	Описание
InitializeSecurityDescriptor	Подготовить новый дескриптор защиты
LookupAccountSid	Найти SID по заданному имени пользователя
SetSecurityDescriptorOwner	Ввести SID владельца в дескриптор защиты
SetSecurityDescriptorGroup	Ввести SID группы в дескриптор защиты
InitializeAcl	Инициализировать DACL или SACL
AddAccessAllowedAce	Добавить к DACL или SACL новый ACE с разрешением доступа
AddAccessDeniedAce	Добавить к DACL или SACL новый ACE с запретом доступа
DeleteAce	Удалить ACE из DACL или SACL
SetSecurityDescriptorDacl	Добавить DACL к дескриптору защиты

Реализация защиты

Защита в автономной системе Windows 2000 реализуется при помощи нескольких компонентов, большую часть которых мы уже рассмотрели (вопросы сетевой безопасности представляют собой совсем другую историю и выходит за рамки данной книги). Регистрацией в системе управляет программа *winlogon*, а аутентификацией занимаются *lsass* и *msgina.dll* (см. раздел «Загрузка Windows 2000» данной главы). Результатом успешной регистрации в системе является новая оболочка с ассоциированным с ней маркером доступа. Этот процесс использует в реестре ключи SECURITY и SAM. Первый ключ определяет общую политику безопасности, а второй ключ содержит информацию о защите для индивидуальных пользователей.

Как только пользователь регистрируется в системе, выполняется операция защиты при открытии объекта. Для каждого вызова OpenXXX требуется имя открываемого объекта и набор прав доступа к нему. Во время обработки процедуры открытия объекта менеджер безопасности (см. рис. 11.2) проверяет наличие у вызывающего процесса соответствующих прав доступа. Для этого он просматривает все маркеры доступа вызывающего процесса, а также список DACL, ассоциированный с объектом. Он просматривает по очереди элементы списка ACL. Как только он находит запись, соответствующую идентификатору SID вызывающего процесса или одной из его групп, поиск прав доступа считается законченным. Если вызывающий процесс обладает необходимыми правами, объект открывается, в противном случае в открытии объекта отказывается.

Помимо разрешающих записей, списки DACL могут также содержать запрещающие записи. Поскольку менеджер безопасности прекращает поиск, наткнувшись на первую запись с указанным идентификатором, запрещающие записи помещаются в начало списка DACL, чтобы пользователь, которому строго запрещен доступ к какому-либо объекту, не смог получить его как член какой-либо группы, которой этот доступ предоставлен.

После того как объект открыт, дескриптор объекта возвращается вызывающему процессу. При последующих обращениях проверяется только, входит ли

данная операция в число операций, разрешенных в момент открытия объекта, чтобы, например, не допустить записи в файл, открытый для чтения.

Кэширование в Windows 2000

В операционной системе Windows 2000, как и в других операционных системах, кэш используется для повышения производительности. Однако устройство менеджера кэша обладает некоторыми необычными свойствами, которые стоит кратко рассмотреть.

Работа менеджера кэша заключается в том, чтобы хранить в памяти недавно использовавшиеся блоки файловой системы и тем самым сокращать время доступа при последующих обращениях к ним. В Windows 2000 есть единый интегрированный кэш, обслуживающий все используемые файловые системы, включая NTFS, FAT-32, FAT-16 и даже файловую систему для CD-ROM. Это означает, что файловые системы не должны управлять собственными кэшами.

В результате менеджер кэша оказывается расположенным в системе в несколько необычном месте (см. рис. 11.2). Он не является частью файловой системы, так как существует несколько независимых файловых систем, у которых может не быть ничего общего. Вместо этого он работает на более высоком уровне, чем файловые системы, формально представляющие собой драйверы, управляемые менеджером ввода-вывода.

В основе организации кэша Windows 2000 лежат не физические блоки, а виртуальные блоки. Чтобы понять, что это означает, вспомним, что традиционные файловые кэши отслеживают блоки с помощью адресов, состоящих из двух частей (раздел, блок), где первое число означает устройство и раздел, а второе число представляет собой номер блока в пределах этого дискового раздела. Менеджер кэша в операционной системе Windows 2000 этого не делает. Вместо этого при обращении к блоку он использует пару (файл, смещение). (Формально кэшируются не файлы, а потоки данных, но мы проигнорируем эту деталь.)

Причина такого неортодоксального устройства заключается в том, что при поступлении запроса к менеджеру кэша в нем указывается пара (файл, смещение), так как это вся информация, доступная вызывающему процессу. Если бы блоки в кэше помечались как (раздел, блок), у менеджера кэша не было бы способа определить, какой блок, заданный парой (файл, смещение), какой паре (раздел, блок) соответствует, так как подобным преобразованием занимается файловая система.

Рассмотрим, как работает менеджер кэша. Когда происходит обращение к файлу, менеджер кэша отображает файл на 256-килобайтный фрагмент виртуального адресного пространства ядра. Если размер файла больше 256 Кбайт, тогда отображается только часть файла. Общий размер виртуального адресного пространства, которое может использовать менеджер кэша, определяется в момент загрузки операционной системы и зависит от объема оперативной памяти. Если менеджеру кэша не хватает 256 Кбайт виртуального адресного пространства, он должен, прежде чем отображать на память новый файл, прекратить отображение на адресное пространство памяти предыдущего файла.

Как только файл отображен на память, менеджер кэша может удовлетворить запросы к его блокам, просто копируя фрагменты памяти из виртуального адресного пространства ядра в буфер пользователя. Если копируемый блок не находится в физической памяти, происходит страничное прерывание, в результате которого менеджер памяти удовлетворяет страничное прерывание обычным способом. Менеджеру кэша даже не известно, был блок в кэше или нет. Операция копирования всегда завершается успешно.

Работа менеджера кэша для случая файловой системы на диске SCSI и файловой системы FAT-32 на диске IDE проиллюстрирована на рис. 11.26. Когда процесс читает файл, запрос направляется менеджеру кэша. Если требуемый блок оказывается в кэше, он немедленно копируется пользователю. Если блока в кэше нет, при попытке копирования блока происходит страничное прерывание. После обработки страничного прерывания блок копируется вызывавшему процессу.

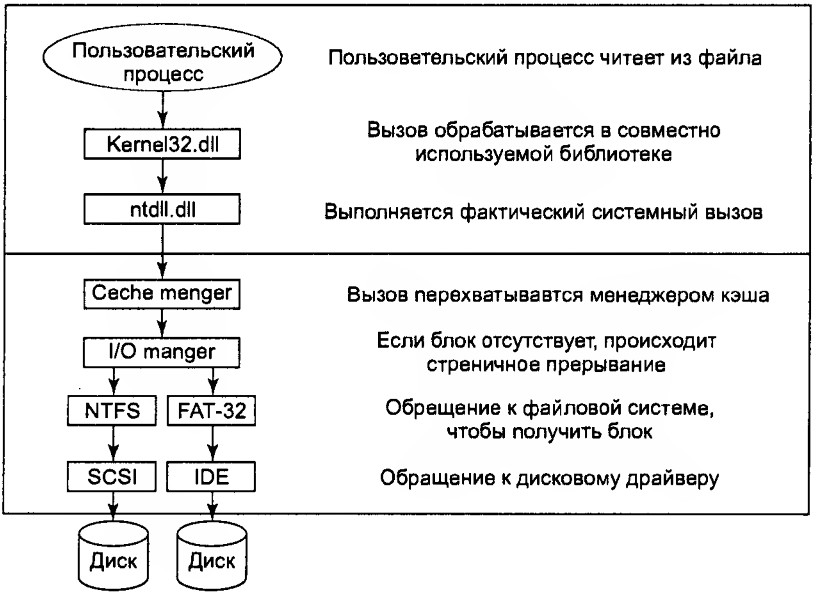


Рис. 11.26. Путь сквозь кэш к апперетуре

В результате применения такой схемы менеджер кэша не знает, сколько из отображаемых им страниц находятся в физической памяти и даже насколько велик кэш. Только менеджер памяти обладает точной информацией. Такой подход позволяет менеджеру памяти динамически менять размер кэша, увеличивая и уменьшая тем самым количество страниц, выделяемых пользовательским процессам. Если обращений к файлам немного, но в памяти компьютера находится много активных процессов, менеджер памяти может отвести большую часть физической памяти под страницы процессов. С другой стороны, если процессов мало, а активность файловой системы высокая, больше физической памяти может быть выделено для кэша.

Другое свойство менеджера кэша состоит в том, что он следит за соответствием файлов, отображенных на память, файлам, открытым для чтения и записи. Рассмотрим, например, ситуацию, в которой один процесс открывает какой-либо файл для чтения и записи, а второй процесс отображает этот же файл на свое адресное пространство. Что произойдет, если второй процесс напрямую запишет данные в этот файл, непосредственно перед тем, как первый процесс прочитает только что измененный блок? Получит ли он устаревшие данные?

В обоих случаях — как для открытых файлов, так и для файлов, отображенных на память, — менеджер кэша отображает на файл 256-килобайтный участок своего виртуального адресного пространства. Файл отображается на память только один раз, независимо от того, сколько процессов откроют его или отобразят на свое адресное пространство. Когда от пользовательского процесса поступает запрос, менеджер кэша просто копирует страницу из памяти в буфер пользовательского процесса. Поэтому копируемая страница всегда отражает текущее состояние файла, так как менеджер кэша использует те же страницы, что и процесс, отображающий файл на память.

Резюме

Структура операционной системы Windows 2000 включает в себя уровень аппаратных абстракций HAL, ядро, исполняющую систему и тонкий уровень системных служб, перехватывающий входящие системные вызовы. Кроме того, операционная система содержит множество драйверов устройств, включая файловую систему и интерфейс графических устройств GDI. Уровень HAL скрывает от верхних уровней определенные различия в аппаратуре. Ядро пытается скрыть от исполняющей системы остальные различия, чтобы сделать ее почти полностью машинно-независимой.

В основе исполняющей системы лежат объекты памяти. Пользовательские процессы создают их и получают дескрипторы, позволяющие управлять этими объектами. Компоненты исполняющей системы также могут создавать объекты. Менеджер объектов управляет пространством имен, в которое могут добавляться объекты для возможности их поиска по имени.

Операционная система Windows 2000 поддерживает процессы, задания, потоки и волокна. У процессов есть виртуальные адресные пространства, кроме того, процессы являются контейнерами ресурсов. Потоки представляют собой единицы исполнения и планируются операционной системой. Волокна являются упрощенными потоками, планируемыми полностью в пространстве пользователя. Задания представляют собой наборы процессов и используются для выделения квот ресурсов. При планировании используется приоритетный алгоритм, в котором управление получает готовый поток с максимальным приоритетом.

Операционной системой Windows 2000 поддерживается виртуальная память с подкачкой по требованию. Алгоритм подкачки основан на понятии рабочего набора. Система управляет несколькими списками свободных страниц, так что когда

происходит страничное прерывание, как правило, у системы уже есть доступная страница. Списки свободных страниц получают страницы, отнимаемые у рабочих наборов при помощи сложных алгоритмов, пытающихся изымать в первую очередь давно не использовавшиеся страницы.

Ввод-вывод осуществляется драйверами устройств, согласующимися с моделью Windows Driver Model. При запуске каждого драйвера инициализируется объект драйвера, содержащий адреса процедур, к которым может обращаться операционная система, чтобы добавить новое устройство или выполнить операцию ввода-вывода. Драйверы могут собираться в стеки или действовать как фильтры.

В основе файловой системы NTFS лежит главная файловая таблица MFT, в которой содержится по одной записи для каждого файла или каталога. У каждого файла есть множество атрибутов, которые могут храниться в записи таблицы MFT или вне ее, на диске. Среди прочих возможностей файловая система NTFS поддерживает сжатие и шифрование.

Защита основывается на списках управления доступом ACL. У каждого процесса есть маркер управления доступом, идентифицирующий процесс, а также указывающий, какими особыми привилегиями он обладает. С каждым объектом ассоциирован дескриптор защиты. Дескриптор защиты указывает на список разграничительного управления доступом DACL, содержащий записи списка ACL, разрешающие или запрещающие доступ к этому объекту отдельным пользователям или группам.

Наконец, операционная система Windows 2000 управляет единым для всех файловых систем кэшем, представляющим собой скорее виртуальный кэш, нежели физический. Запросы к дисковым блокам сначала поступают в кэш. Если нужных блоков там нет, тогда вызывается соответствующая файловая система.

Вопросы

1. Назовите одно преимущество и один недостаток использования реестра вместо отдельных файлов *.ini*.
2. У мыши может быть 1, 2 или 3 кнопки. Используются все три типа мыши. Скрывает ли эти различия уровень HAL от остальной части операционной системы? Почему да или почему нет?
3. Уровень HAL поддерживает дату, начиная от года 1601. Приведите пример приложения, в котором это свойство полезно.
4. Подсистема POSIX нужна для реализации сигналов в стиле UNIX. Если пользователь нажмет клавишу для сигнала Quit, будет ли он планироваться как DPC или APC?
5. Многие компоненты исполняющей системы (см. рис. 11.2) обращаются к другим компонентам исполняющей системы. Приведите три примера вызова одним компонентом другого компонента, но используйте различные компоненты (шесть компонентов).

6. В интерфейсе Win32 нет сигналов. Если бы сигналы были введены, они могли бы относиться к процессам, потокам, и к тем и к другим, ни к тем, ни к другим. Выскажите свое предложение, к чему их отнести, и объясните, чем хороша ваша идея.
7. Альтернатива использования DLL заключается в том, чтобы статически компоновать каждую программу точно с теми библиотечными процедурами, к которым она обращается. Если бы было нужно внедрить эту схему, где бы это имело больший смысл, на клиентах или на серверах?
8. Файл *ntdll.dll* экспортирует 1179 функциональных вызовов, тогда как файл *ntoskrnl.exe* экспортирует 1209 функциональных вызовов. Является ли это ошибкой? Чем может быть вызвано это различие?
9. Объекты, управляемые менеджером объектов, имеют переменный размер, у различных объектов могут быть разные размеры. Может ли объект начинаться с произвольного байта в невыгружаемом пуле? *Подсказка*: для ответа на этот вопрос не требуется дополнительной информации об операционной системе Windows 2000, кроме той, что была дана в тексте.
10. Существует ли ограничение на число различных операций, которые могут быть определены для исполняющего объекта? Если да, то какова природа этого ограничения? Если нет, то почему?
11. Вызов интерфейса Win32 `WaitForMultipleObjects` позволяет потоку блокироваться на множестве объектов синхронизации, чьи дескрипторы передаются этой функции в виде параметров. Вызывающий поток отпускается, как только один из этих объектов получает сигнал. Может ли набор объектов синхронизации включать в себя два семафора, один мьютекс и одну критическую область? Почему да или почему нет? *Подсказка*: этот вопрос требует тщательного обдумывания.
12. Назовите три причины, по которым процесс может завершиться.
13. Рассмотрите ситуацию на рис. 11.8, в которой система собирается планировать поток. Если предположить, что каждый поток ограничен по скорости вычислений, то сколько времени пройдет, прежде чем поток с приоритетом 3 получит управление в операционной системе Windows 2000 Professional?
14. Допустим, что квант времени установлен равным 20 мс и что текущий поток с приоритетом 24 только что начал свой квант. Внезапно операция ввода-вывода завершается, и поток с приоритетом 28 переходит в состояние готовности. Сколько времени придется ему ждать обслуживания?
15. В операционной системе Windows 2000 текущий приоритет всегда больше базового приоритета или равен ему. Существуют ли обстоятельства, при которых имел бы смысл приоритет ниже базового? Если да, то приведите пример. Если нет, то почему нет?
16. Некоторые программы для операционной системы MS-DOS были написаны на ассемблере, с использованием таких команд, как `IN` и `OUT`, для обращения к портам. Может ли такая программа выполняться в системе Windows 2000? Если нет, то можете ли вы придумать способ поддержки подобных программ?

17. Назовите два способа получения лучшего времени отклика для важных процессов.
18. В тексте главы кратко обсуждался позиционно-независимый код. Эта технология применяется для того, чтобы два процесса могли совместно использовать одну и ту же процедуру в различных виртуальных адресах. Можно ли решить эту проблему при помощи простой настройки таблиц страниц двух процессов? Аргументируйте свой ответ.
19. В операционной системе Windows 2000 совместно используемые библиотеки хранятся в файлах *.dll*, которые могут одновременно отображаться различными процессами на свои адресные пространства. Проблема может возникнуть, когда два процесса захотят отобразить одну и ту же библиотеку по различным виртуальным адресам. Как может быть решена эта проблема на компьютере с процессором Pentium, учитывая свойство архитектуры его памяти? Если это решение требует изменений в реализации Windows 2000, перечислите эти изменения.
20. Если область виртуального адресного пространства зарезервирована, но не фиксирована, как вы думаете, создается ли для нее описатель виртуальной памяти VAD? Аргументируйте свой ответ.
21. Какие из переходов, показанных на рис. 11.14, представляют собой политические решения, в отличие от переходов, вынуждаемых системными событиями (например, процесс завершает работу, и его страницы освобождаются)?
22. Предположим, страница используется совместно одновременно в двух рабочих наборах. Куда она попадает (см. рис. 11.14), если удаляется из одного из рабочих наборов?
23. Когда процесс освобождает чистую страницу, он выполняет переход (5) на рис. 11.14. Почему нет перехода в список модифицированных страниц, когда освобождается «грязная» страница?
24. Допустим, что 32-разрядное слово может быть прочитано или записано за 10 нс в оперативную память или видеопамять. Сколько времени потребуется, чтобы нарисовать фон для экрана XGA в лучшем случае?
25. Файл размещается на диске следующим образом. Какими будут записи серий блоков в таблице MFT?

    Offset 0 1 2 3 4 5 6 7 8 9 10

    Disk address 50 51 52 22 24 25 26 53 54 – 60
26. Рассмотрите запись таблицы MFT на рис. 11.18. Предположим, файл вырос и к концу файла был добавлен 10-й блок. Номер этого блока 66. Как теперь будет выглядеть запись MFT?
27. На рис. 11.22, *б* первые два сегмента файла имеют длину по 8 блоков. Является ли равенство их размеров случайностью, или же это объясняется свойствами работы алгоритма сжатия? Аргументируйте свой ответ.

28. Допустим, вы решили создать облегченную версию операционной системы Windows 2000. Какие поля из перечисленных на рис. 11.24 можно удалить, не ослабляя защиты системы?

29. Во всех текущих версиях Windows с помощью команды *regedit* можно экспортировать часть или весь реестр в текстовый файл. Сохраните реестр в виде текстового файла несколько раз во время работы и посмотрите, какие изменения в нем обнаружатся. Если у вас есть доступ к компьютеру с операционной системой Windows, на котором вы можете устанавливать программное или аппаратное обеспечение, определите, какие изменения появятся в реестре при добавлении или удалении программы или устройства.

30. Напишите программу для операционной системы UNIX, имитирующую запись в файл NTFS с несколькими потоками данных. В качестве аргументов эта программа должна принимать список из одного или нескольких файлов. Эта программа должна создавать выходной файл, в котором один поток должен содержать атрибуты всех аргументов, а дополнительные потоки — содержимое каждого из аргументов. Затем напишите вторую программу, сообщающую об атрибутах и потоках и извлекающую все компоненты.

31. Напишите программу, формирующую имя формата MS-DOS 8+3 по заданному имени файла. Используйте алгоритм Windows 2000.

Глава 12 Разработка операционных систем

В предшествующих 11 главах мы рассмотрели большое количество материала и познакомились со множеством понятий и примеров, имеющих отношение к операционным системам. Однако разработка новой операционной системы отличается от изучения существующих операционных систем. В этой главе мы собираемся обсудить несколько вопросов, которые должны учитывать разработчики новых операционных систем.

В среде разработчиков операционных систем ходит множество изустных преданий о том, что такое хорошо и что такое плохо, однако на удивление малое количество из этих историй записано. Наиболее важной книгой можно назвать классический труд Фреда Брукса [44], в котором автор делится своим опытом проектирования и реализации операционной системы IBM OS/360. Материал выпущенного к 20-летней годовщине издания был пересмотрен, к тому же в содержание книги было включено несколько новых глав [46]. Вероятно, единственной книгой по операционным системам, в которой серьезно обсуждается тема проектирования, является [80].

Тремя классическими трудами по проектированию операционных систем являются [194, 74, 286]. Как и книги Брукса, эти три статьи успешно пережили время, прошедшее с момента их написания. Большая часть рассматриваемых в них вопросов сохранила свою актуальность и в наши дни.

Данная глава основана на содержимом этих источников, кроме того, в ней используется личный опыт участия автора в проектировании трех систем: Amoeba [324], MINIX [326] и Globe [340]. Поскольку среди разработчиков операционных систем нет единого мнения по вопросу о том, как лучше всего проектировать операционные системы, эта глава будет носить более личный характер, более умозрительный и, несомненно, более противоречивый, чем предыдущие главы.

Природа проблемы проектирования

Разработка операционных систем представляет собой в большей мере инженерный проект, нежели точную науку. В этой области значительно труднее наметить ясные цели и достичь их. Рассмотрим для начала вопрос постановки задачи.

Цели

Чтобы проект операционной системы был успешным, разработчики должны иметь четкое представление о том, чего они хотят. При отсутствии цели очень трудно принимать последующие решения. Чтобы этот вопрос стал понятнее, полезно взглянуть на два языка программирования, PL/1 и C. Язык PL/1 был разработан корпорацией IBM в 60-е годы, так как поддерживать одновременно FORTRAN и COBOL и слушать при этом за спиной ворчание ученых о том, что Algol лучше, чем FORTRAN и COBOL вместе взятые, было невыносимо. Поэтому был создан комитет для создания нового языка, удовлетворяющего запросам всех программистов: PL/1. Этот язык обладал некоторыми чертами языка FORTRAN, некоторыми особенностями языка COBOL и некоторыми свойствами языка Algol. Проект потерпел неудачу, потому что ему недоставало единой концепции. Проект представлял собой лишь набор свойств, конфликтующих друг с другом, к тому же язык PL/1 был слишком громоздким и неуклюжим, чтобы программы на нем можно было эффективно компилировать.

Теперь взглянем на язык C. Он был спроектирован всего одним человеком (Деннисом Ритчи) для единственной цели (системного программирования). Успех его был колоссален, и это не в последнюю очередь объяснялось тем, что Ритчи знал, чего хотел и чего не хотел. В результате спустя десятилетия после своего появления этот язык все еще широко распространен. Наличие четкого представления о своих целях является решающим.

Чего же хотят разработчики операционных систем? Очевидно, ответ варьируется от системы к системе и будет разным для встроенных систем и серверных систем. Для универсальных операционных систем основными являются следующие четыре пункта:

1. Определение абстракций.
2. Предоставление примитивных операций.
3. Обеспечение изоляции.
4. Управление аппаратурой.

Ниже будет рассмотрен каждый из этих пунктов.

Наиболее важная, но, вероятно, наиболее сложная задача операционной системы заключается в определении правильных абстракций. Некоторые из них, такие как процессы и файлы, уже используются так давно, что могут показаться очевидными. Другие, такие как потоки исполнения, представляют собой более новые и потому не столь устоявшиеся понятия. Например, если состоящий из нескольких потоков процесс, один из потоков которого блокирован вводом с клавиатуры, клонируется, то должен ли поток в новом процессе также ожидать ввода с клавиатуры? Другие абстракции относятся к синхронизации, сигналам, модели памяти, моделированию ввода-вывода и иным областям.

Каждая абстракция может быть реализована в виде конкретных структур данных. Пользователи могут создавать процессы, файлы, семафоры и т. д. Управляют этими структурами данных при помощи примитивных операций. Например, пользователи могут читать и писать файлы. Примитивные операции реализуются в виде системных вызовов. С точки зрения пользователя, сердце операционной

системы формируется абстракциями, а операции над ними возможны при помощи системных вызовов.

Поскольку на одном компьютере могут одновременно зарегистрироваться несколько пользователей, операционная система должна предоставлять механизмы для отделения их друг от друга. Один пользователь не должен вмешиваться в работу другого. Для группирования ресурсов с целью их защиты широко применяется концепция процессов. Как правило, также защищаются файлы и другие структуры данных. Ключевая цель проектирования операционной системы заключается в том, чтобы гарантировать, что каждый пользователь может выполнять только разрешенные ему действия с данными, к которым у него есть право доступа. Однако пользователям также бывает необходимо совместное использование данных и ресурсов, поэтому изоляция должна быть избирательной и контролироваться пользователями. Все это существенно усложняет устройство операционной системы.

С этим вопросом тесно связана проблема необходимости изолирования отказов. Если какая-либо часть системы выйдет из строя (чаще всего это один из пользовательских процессов), сбойный процесс не должен нарушить работу всей операционной системы. Устройство операционной системы должно гарантировать надежную изоляцию различных частей операционной системы друг от друга.

Наконец, операционная система должна управлять аппаратурой. В частности, она должна заботиться обо всех низкоуровневых микросхемах, таких как контроллеры прерываний и контроллеры шин. Она также должна обеспечивать каркас для того, чтобы драйверы устройств могли управлять крупными устройствами ввода-вывода, такими как диски, принтеры и дисплей.

Почему так сложно спроектировать операционную систему?

Закон Мура гласит, что аппаратное обеспечение компьютера увеличивает свою производительность в 100 раз каждые десять лет. Никто, к сожалению, так и не сформулировал ничего подобного для программного обеспечения. Никто и не говорит, что операционные системы вообще совершенствуются с годами. В самом деле, можно утверждать, что некоторые из них даже стали хуже в определенных аспектах (таких как надежность), чем, например, операционная система UNIX Version 7 образца 70-х годов.

Почему? Как правило, основными причинами оказываются инерция и желание сохранить обратную совместимость, хотя неумение разработчиков придерживаться принципов хорошего проектирования тоже вносит свою лепту. Но этот вопрос следует обсудить подробнее. Операционные системы принципиально отличаются от небольших прикладных программ, продающихся в компьютерных магазинах за 49 долларов. Рассмотрим восемь аспектов, благодаря которым разработка операционной системы оказывается значительно сложнее написания прикладной программы.

Во-первых, операционные системы стали крайне громоздкими программами. Никто в одиночку не может, засев за персональный компьютер на несколько месяцев, написать операционную систему. Все современные версии UNIX превосходят 1 млн строк исходного текста. Операционная система Windows 2000 состоит

из 29 млн строк кода. Ни один человек на Земле не способен понять даже одного миллиона строк, не говоря уже о 29 миллионах. Если вы создаете продукт, который ни один из разработчиков не может даже надеяться понять целиком, не удивительно, что результаты часто оказываются далеки от оптимальных.

Операционные системы не являются самыми сложными системами в мире. Например, авианосцы представляют собой значительно более сложные системы, но они легче разделяются на отдельные подсистемы. Проектировщики туалетов на авианосце не должны заниматься радарной системой. Эти две подсистемы почти не взаимодействуют. В операционной системе файловая система часто взаимодействует с системой памяти самым неожиданным и непредсказуемым образом.

Во-вторых, операционные системы должны иметь дело с параллелизмом. В системе одновременно присутствует множество пользователей, работающих с множеством устройств ввода-вывода. Управление несколькими параллельно выполняющимися процессами существенно сложнее управления одной последовательной деятельностью. Среди множества возникающих при этом проблем достаточно назвать хотя бы состязания и тупиковые ситуации.

В-третьих, операционные системы должны учитывать наличие потенциально враждебных пользователей — пользователей, желающих вмешиваться в работу операционной системы или выполнять запрещенные действия, например похищение чужих файлов. Операционная система должна предпринимать меры для предотвращения подобных действий со стороны злонамеренных пользователей. Текстовые процессоры и фоторедакторы не сталкиваются с подобными проблемами.

В-четвертых, несмотря на тот факт, что пользователи друг другу не доверяют, многим из них бывает нужно использовать какую-либо информацию или ресурсы совместно с определенной группой пользователей. Операционная система должна предоставлять эту возможность, но таким образом, чтобы злоумышленник не смог воспользоваться этими свойствами для своих целей. У прикладных программ подобных проблем тоже нет.

В-пятых, операционные системы, как правило, живут довольно долгое время. Операционная система UNIX использовалась в течение четверти века; система Windows уже используется более десяти лет и признаков умирания не обнаруживает. Соответственно, проектировщики операционной системы должны думать о том, как могут измениться аппаратура и приложения в отдаленном будущем и как им следует к этому подготовиться. Системы, отражающие одно специфическое мировоззрение, как правило, быстро сходят со сцены.

В-шестых, у разработчиков операционной системы на самом деле нет четкого представления о том, как будет использоваться их система, поэтому они должны обеспечить достаточную степень универсальности. При проектировании таких систем, как UNIX или Windows 2000, не предполагалось их использование для работы с электронной почтой или запуск web-браузера под их управлением, однако многие современные компьютеры в основном только для этого и используются. Трудно себе представить проектировщика морского судна, который не знал бы, что он проектирует: рыболовецкое судно, пассажирское судно или военный корабль.

В-седьмых, от современных операционных систем, как правило, требуется переносимость, что означает возможность работы на различных платформах. Они также должны поддерживать сотни или даже тысячи устройств ввода-вывода,

которые проектируются совершенно независимо друг от друга. Например, операционная система должна работать на компьютерах как с прямым, так и с обратным порядком байтов. Второй пример постоянно наблюдался в системе MS-DOS, когда пользователи пытались установить звуковую карту и модем, использующие одни и те же порты ввода-вывода или линии запроса прерывания. С такими проблемами, как конфликты различных частей аппаратуры, приходится иметь дело в основном именно операционным системам.

Наконец, в-восьмых, при разработке операционных систем часто учитывается необходимость совместимости с предыдущей версией операционной системы. Система может иметь множество ограничений на длину слов, имена файлов и т. д., рассматриваемых теперь проектировщиками как устаревшие, но от которых трудно избавиться. Это напоминает переоборудование автомобильного завода под выпуск новых моделей с условием сохранения текущих мощностей по выпуску старых моделей.

Разработка интерфейса

Итак, теперь должно быть ясно, что написание современной операционной системы представляет собой непростую задачу. Но с чего начинается эта работа? Возможно, лучше всего сначала подумать о предоставляемых операционной системой интерфейсах. Операционная система предоставляет набор служб, большую часть типов данных (например, файлы) и множество операций с ними (например, read). Вместе они формируют интерфейс для пользователей системы. Обратите внимание, что в данном контексте пользователями операционной системы являются программисты, пишущие программы, которые используют системные вызовы, а не люди, запускающие прикладные программы.

Кроме основного интерфейса системных вызовов, у большинства операционных систем есть дополнительные интерфейсы. Например, некоторым программистам бывает необходимо написать драйвер устройства, чтобы добавить его в операционную систему. Эти драйверы предоставляют определенные функции и могут обращаться к определенным системным вызовам. Функции и вызовы определяют интерфейс, существенно отличающийся от используемого прикладными программистами. Все эти интерфейсы должны быть тщательно спроектированы, если разработчики системы рассчитывают на успех.

Руководящие принципы

Существуют ли принципы, руководствуясь которыми можно проектировать интерфейсы? Мы надеемся, что такие принципы есть. Если выразить их в нескольких словах, то это простота, полнота и возможность эффективной реализации.

Принцип 1. Простота

Простой интерфейс легче понять и реализовать без ошибок. Всем разработчикам систем следует помнить эту знаменитую цитату французского летчика и писателя Антуана де Сент-Экзюпери:

Совершенство достигается не тогда, когда уже больше нечего добавить, а когда больше нечего убавить.

Этот принцип утверждает, что лучше меньше, чем больше, по крайней мере, применительно к операционным системам. Другими словами, этот принцип может быть выражен следующей аббревиатурой, предлагающей программисту, в чьих мыслительных способностях возникают сомнения, не усложнять систему: KISS (Keep It Simple, Stupid).

Принцип 2. Полнота

Разумеется, интерфейс должен предоставлять пользователям возможность выполнять все, что им необходимо, то есть интерфейс должен обладать полнотой. В связи с этим на ум приходит другая цитата, на этот раз это фраза, сказанная Альбертом Эйнштейном:

Все должно быть простым, насколько это возможно, но не проще.

Другими словами, операционная система должна выполнять то, что от нее требуется, но не более того. Если пользователю нужно хранить данные, операционная система должна предоставлять для этого некий механизм. Если пользователям необходимо общаться друг с другом, операционная система должна предоставлять механизм общения и т. д. В своей речи по поводу получения награды Turing Award один из разработчиков систем CTSS и MULTICS Фернандо Корбато объединил понятия простоты и полноты и сказал:

Во-первых, важно подчеркнуть значение простоты и элегантности, так как сложность приводит к нагромождению противоречий и, как мы уже видели, появлению ошибок. Я бы определил элегантность как достижение заданной функциональности при помощи минимума механизма и максимума ясности.

Ключевая идея здесь — *минимум механизма*. Другими словами, каждая функция, каждый системный вызов должны нести свою собственную ношу. Когда член команды проектировщиков предлагает расширить системный вызов или добавить новую функцию, остальные разработчики должны спросить его: «Произойдет ли что-нибудь ужасное, если мы этого не сделаем?» Если ответом будет: «Нет, но, возможно, когда-нибудь кто-то посчитает эту функцию полезной», поместите ее в библиотеку уровня пользователя, а не в операционную систему, даже если при этом она будет работать медленнее. Не должна ставиться задача сделать каждую функцию быстрее пули. Цель заключается в том, чтобы сохранить то, что Корбато назвал минимумом механизма.

Давайте кратко рассмотрим два примера из моего личного опыта: операционные системы MINIX и Amoeba. Для всех задач в системе MINIX есть три системных вызова: `send`, `receive` и `sendrec`. Система структурирована в виде набора процессов, с менеджером памяти, файловой системой и каждым драйвером устройства, представляющими собой отдельный планируемый процесс. В первом приближении все, что делает ядро, — это планирование процессов и управление передачей сообщений между ними. Соответственно, необходимыми являются только два системных вызова: `send` для отправки сообщения и `receive`, чтобы сообщение получить. Третий

системный вызов, sendrec, представляет собой просто оптимизацию, позволяющую послать сообщение и запросить ответ всего за одно эмулированное прерывание. Все остальное в системе выполняется с помощью обращения к какому-либо другому процессу (например, к процессу файловой системы или к дисковому драйверу).

Операционная система Amoeba устроена еще проще. У нее есть только один системный вызов: выполнение вызова удаленной процедуры. Этот вызов отправляет сообщение и ждет ответа. По существу, это то же самое, что и системный вызов sendrec в системе MINIX. Все остальное строится на этом единственном вызове.

Принцип 3. Эффективность

Третий руководящий принцип представляет собой эффективность реализации. Если какая-либо функция (или системный вызов) не может быть реализована эффективно, вероятно, ее не следует реализовывать. Программист должен интуитивно представлять стоимость реализации и использования каждого системного вызова. Например, программисты, пишущие программы в системе UNIX, считают, что системный вызов lseek дешевле системного вызова read, так как первый системный вызов просто изменяет содержимое указателя, хранящегося в памяти, тогда как второй системный вызов выполняет операцию дискового ввода-вывода. Если интуиция подводит программиста, программист создает неэффективные программы.

Парадигмы

Когда цели установлены, можно начинать проектирование. Можно начать, например, со следующего: подумать, как будет представать система перед пользователями. Один из наиболее важных вопросов заключается в том, чтобы все функции системы хорошо согласовывались друг с другом и обладали тем, что часто называют **архитектурной согласованностью**. При этом важно различать два типа «пользователей» операционной системы. С одной стороны, существуют *пользователи*, взаимодействующие с прикладными программами; с другой стороны, есть *программисты*, пишущие эти прикладные программы. Первые большей частью имеют дело с графическим интерфейсом пользователя, тогда как последние в основном взаимодействуют с интерфейсом системных вызовов. Если задача заключается в том, чтобы иметь единый графический интерфейс пользователя, заполняющий всю систему, как, например, в системе Macintosh, тогда разработку следует начать отсюда. Если же цель состоит в том, чтобы обеспечить поддержку различных возможных графических интерфейсов пользователя, как в системе UNIX, тогда в первую очередь должен быть разработан интерфейс системных вызовов. Начало разработки системы с графического интерфейса пользователя представляет собой, по сути, проектирование сверху вниз. Вопрос заключается в том, какие функции будет этот интерфейс иметь, как будет пользователь с ними взаимодействовать и как следует спроектировать систему для их поддержки. Например, если большинство программ отображает на экране значки, а затем ждет, когда пользователь щелкнет на них мышью, это предполагает использование управляемой событиями модели для графического интерфейса пользователя и, возможно, для операционной системы. С другой стороны, если экран в основном заполнен текстовыми окнами, тогда,

вероятно, лучшей представляется модель, в которой процессы считывают символы с клавиатуры.

Реализация в первую очередь интерфейса системных вызовов представляет собой проектирование снизу вверх. Здесь вопросы заключаются в том, какие функции нужны программистам. В действительности для поддержки графического интерфейса пользователя требуется не так уж много специальных функций. Например, оконная система под названием X Windows, используемая в UNIX, представляет собой просто большую программу на языке C, которая обращается к клавиатуре, мыши и экрану с системными вызовами `read` и `write`. Оконная система X Windows была разработана значительно позже операционной системы UNIX, но для ее работы не потребовалось большого количества изменений в операционной системе. Это подтверждает тот факт, что система UNIX обладает полнотой в достаточной степени.

Парадигмы интерфейса пользователя

Как для интерфейса уровня графического интерфейса пользователя, так и для интерфейса системных вызовов наиболее важный аспект заключается в наличии хорошей парадигмы (иногда называемой метафорой или модельным представлением), обеспечивающей наглядный зрительный образ интерфейса. Многими графическими интерфейсами пользователя используется парадигма WIMP (Windows, Icons, Menus, Pointers — окна, пиктограммы, меню, указатели), обсуждавшаяся в главе 5. Эта парадигма используется в интерфейсе для обеспечения согласованности таких идиом, как щелчок и двойной щелчок мышью, перетаскивание и т. д. Часто к программам применяются дополнительные требования, такие как наличие строки меню с пунктами **Файл**, **Правка** и т. д., каждый из которых содержит хорошо знакомые пункты меню. Таким образом, пользователи, знакомые с одной программой, легко могут освоить другую программу.

Однако пользовательский интерфейс WIMP не является единственно возможным. В некоторых карманных компьютерах применяется интерфейс электронного пера. Устройства, предназначенные для мультимедиа, могут использовать интерфейс видеомагнитофона. И, разумеется, при управлении компьютером голосом используется совершенно отличная парадигма. Выбор парадигмы, конечно, важен, но еще важнее сам факт использования единой парадигмы, объединяющей весь пользовательский интерфейс.

Какая бы парадигма ни была выбрана, существенно, чтобы все прикладные программы использовали ее. Следовательно, проектировщики системы должны предоставить разработчикам прикладных программ библиотеки и инструменты для доступа к процедурам, обеспечивающим однородный внешний вид пользовательского интерфейса. Разработка пользовательского интерфейса представляет собой очень важную задачу, но она не является темой данной книги, поэтому мы вернемся к обсуждению темы интерфейса операционной системы.

Парадигмы исполнения

Архитектурная согласованность важна на уровне пользователя, но в равной мере она важна на уровне интерфейса системных вызовов. Здесь часто полезно различать парадигму исполнения от парадигмы данных, поэтому мы рассмотрим и ту и другую, начав с первой.

Широкое распространение получили две парадигмы: алгоритмическая и движимая событиями. **Алгоритмическая парадигма** основана на идее, что программа запускается для выполнения некоторой функции, известной заранее или задаваемой в виде параметров. Эта функция может заключаться в компиляции программы, составлении ведомости или пилотировании самолета до Сан-Франциско. Базовая логика жестко прошита в код программы, при этом программа время от времени обращается к системным вызовам, чтобы получить ввод пользователя, обратиться к системным службам и т. д. Этот подход проиллюстрирован в листинге 12.1, *а*.

Другая парадигма исполнения представляет собой **парадигму управления событиями** (листинг 12.1, *б*). Здесь программа выполняет определенную инициализацию, например, отображает какое-либо окно, а затем ждет, когда операционная система сообщит ей о первом событии. Этим событием может быть нажатие клавиши или перемещение мыши. Такая схема полезна для программ, активно взаимодействующих с пользователем.

Листинг 12.1. Алгоритмический код (*а*); движимый событиями код (*б*)

```
main()
{
    int ... ;

    init();
    do_something();
    read(...);
    do_something_else();
    write(...);
    keep_going();
    exit(0);
}
```

а

```
main()
{
    mess_t msg;

    init();
    while (get_message(&msg)) {
        switch (msg.type) {
            case 1: ... ;
            case 2: ... ;
            case 3: ... ;
        }
    }
}
```

б

Каждая парадигма порождает собственный стиль программирования. В алгоритмической парадигме алгоритмы занимают центральное положение, а операционная система рассматривается как поставщик служб. В парадигме управления событиями операционная система также предоставляет службы, но ее основная роль заключается в координации активности пользователя и формировании событий, потребляемых процессами.

Парадигмы данных

Парадигма исполнения является не единственной парадигмой, экспортируемой операционной системой. Не менее важна парадигма данных. Ключевой вопрос здесь заключается в том, как предстают перед программистом системные структуры и устройства. В ранних пакетных системах, предназначенных для выполнения программ на языке FORTRAN, все моделировалось как логическая магнитная лента. Считываемые колоды карт воспринимались как входные ленты, пробиваемые колоды карт обрабатывались как выходные ленты. Вывод на принтер также обрабатывался как выходная лента. Файлы на диске также считались лентами. Произвольный доступ к файлу был возможен только при помощи перемотки соответствующей ленты и повторного считывания.

Задание запускалось при помощи карт управления заданием, например:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

Первая карта представляла собой инструкцию для оператора. Он должен был достать из шкафа бобину номер 781 и установить ее на накопителе 8. Вторая карта являлась командой операционной системе запустить только что откомпилированную с языка FORTRAN программу, отображая *INPUT* (означающий устройство чтения перфокарт) на логическую ленту 1, дисковый файл *MYDATA* на логическую ленту 2, принтер *OUTPUT* на логическую ленту 3, перфоратор *PUNCH* на логическую ленту 4 и физический накопитель на магнитной ленте *TAPE08* на логическую ленту 5.

Синтаксис языка FORTRAN позволяет читать и писать логические ленты. При чтении с логической ленты 1 программа получает ввод с перфокарт. При помощи записи на логическую ленту 3 программа может вывести результаты на принтер. Обращаясь к логической ленте 5, программа может читать и писать магнитную ленту 781 и т. д. Обратите внимание, что идея магнитной ленты представляла собой всего лишь парадигму (модель) для объединения устройства чтения перфокарт, принтера, перфоратора, дисковых файлов и магнитофонов. В данном примере только логическая лента 5 была физической лентой. Все остальное представляло собой обычные файлы для подкачки данных. Это была примитивная парадигма, но она была шагом в правильном направлении.

Затем появилась операционная система UNIX, которая пошла значительно дальше в этом направлении, используя модель «все суть файлы». При использовании этой парадигмы все устройства ввода-вывода рассматриваются как файлы, которые можно открывать и управлять ими, как обычными файлами. Операторы на языке C

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

открывают настоящий дисковый файл и терминал пользователя. Последующие операторы могут использовать дескрипторы файлов *fd1* и *fd2*, чтобы читать из этих файлов и писать в них. С этого момента нет разницы между доступом к файлу и доступом к терминалу, не считая того, что при обращении к терминалу не разрешается операция перемещения указателя в файле.

Операционная система UNIX не только объединяет файлы и устройства ввода-вывода, но также позволяет получать доступ к другим процессам через каналы, как к файлам. Более того, если поддерживается отображение файлов на адресное пространство памяти, процесс может обращаться к своей виртуальной памяти так, как если бы это был файл. Наконец, в версиях UNIX, поддерживающих файловую систему */proc*, строка на языке C

```
fd3 = open("/proc/501", O_RDWR);
```

позволяет процессу (попытаться) получить доступ к памяти процесса 501 для чтения и записи при помощи дескриптора файла *fd3*, что может быть полезно, например, при отладке программы.

Операционная система Windows 2000 идет в использовании этой модели еще дальше, представляя все, что есть в системе, в виде объектов. Получив дескриптор

файла, процесса, семафора, почтового ящика или другого объекта ядра, процесс может выполнять с этим объектом различные действия. Эта парадигма является еще более общей, чем используемая в UNIX, и значительно более общей, чем в FORTRAN.

Объединяющие парадигмы также встречаются в других контекстах. Следует отметить один из них: Всемирную паутину (Web). Используемая в Паутине парадигма состоит в том, что все киберпространство заполнено документами, у каждого их которых есть свой адрес URL. Обратившись по соответствующему указателю URL (введя его с клавиатуры или щелкнув мышью по ссылке), вы получаете этот документ. В действительности многие «документы» вовсе не являются документами, но формируются программой или сценарием оболочки, когда поступает запрос. Например, когда пользователь запрашивает в Инетернет-магазине список компакт-дисков конкретного исполнителя, документ создается на лету программой. Он совершенно точно не существовал до того, как был получен запрос.

Итак, мы рассмотрели четыре парадигмы, а именно: все суть ленты, файлы, объекты или документы. Во всех четырех случаях задача заключается в том, чтобы унифицировать данные, устройства или другие ресурсы для упрощения работы с ними. Каждая операционная система должна иметь подобную унифицирующую парадигму данных.

Интерфейс системных вызовов

Если исходить из высказанного Корбато принципа минимального механизма, тогда операционная система должна предоставлять настолько мало системных вызовов, насколько это возможно (необходимый минимум), и каждый системный вызов должен быть настолько прост, насколько это возможно (но не проще). Объединяющая парадигма данных может играть главную роль в этом. Например, если файлы, процессы, устройства ввода-вывода и прочее будут выглядеть как файлы или объекты, все они могут читаться при помощи всего одного системного вызова `read`. В противном случае пришлось бы иметь различные системные вызовы, такие как `read_file`, `read_proc`, `read_tty` и т. д.

В некоторых случаях может потребоваться несколько вариантов системных вызовов, но, как правило, на практике лучше иметь один системный вызов, обрабатывающий общий случай, с различными библиотечными процедурами, скрывающими этот факт от программистов. Например, в операционной системе UNIX есть системный вызов `exec` для замены виртуального адресного пространства процесса. Наиболее общий вариант его использования выглядит следующим образом:

```
exec(name. argp. envp);
```

Данный системный вызов загружает исполняемый файл *name* и передает ему аргументы, на которые указывает *argp*, и список переменных окружения, на который указывает *envp*. Иногда бывает удобнее явно перечислить аргументы, поэтому в библиотеках содержатся процедуры, вызываемые следующим образом:

```
execl(name. arg0. arg1. .... argn. 0);
execle(name. arg0. arg1. .... argn. envp);
```

Эти процедуры всего лишь помещают аргументы в массив, после чего обращаются к системному вызову exec, который и выполняет всю работу. Такая схема является лучшей в обоих смыслах: благодаря единственному простому системному вызову операционная система сохраняет свою простоту, в то же самое время программист получает возможность обращаться к системному вызову exec различными способами.

Разумеется, пытаясь использовать один-единственный системный вызов для всех случаев жизни, легко дойти до крайностей. Для создания процесса в операционной системе UNIX требуется два системных вызова: fork, за которым следует exec. У первого вызова нет параметров; у второго вызова три параметра. Для сравнения: у вызова Win32 API для создания процесса, CreateProcess, 10 параметров, один из которых представляет собой указатель на структуру с дополнительными 18 параметрами.

Давным-давно следовало задать вопрос: «Произойдет ли катастрофа, если мы опустим что-нибудь из этого?» Правдивый ответ должен был звучать так: «В некоторых случаях программист будет вынужден совершить больше работы для достижения определенного эффекта, зато мы получим более простую, компактную и надежную операционную систему». Конечно, человек, предлагающий версию с этими 10 + 18 параметрами, мог добавить: «Но пользователи любят все эти возможности». Возразить на это можно было бы так: «Еще больше им нравятся системы, которые используют мало памяти и никогда не ломаются». Компромисс, заключающийся в большей функциональности за счет использования большего объема памяти, по крайне мере, виден невооруженным глазом и ему можно дать оценку (так как стоимость памяти известна). Однако трудно оценить количество дополнительных сбоев в год, которые появятся благодаря внедрению новой функции. Кроме того, неизвестно, сделали бы пользователи тот же выбор, если им заранее была известна эта скрытая цена. Этот эффект можно резюмировать первым законом программного обеспечения Таненбаума:

При увеличении размера программы количество содержащихся в ней ошибок также увеличивается.

Когда к программе добавляются новые функции, при этом к ней добавляются новые процедуры, а вместе с ними и новые ошибки. Программисты, полагающие, что при добавлении новых функций к программе не добавится новых ошибок, либо являются новичками в программировании, либо верят, что за ними присматривает добрая фея.

Простота является не единственным принципом, которым следует руководствоваться при разработке системных вызовов. Следует также помнить о фразе, сказанной Б. Лэмпсоном в 1984 году:

Не скрывай мощь.

Если у аппаратного обеспечения есть крайне эффективный способ выполнить что-либо, программистам следует предоставить простой доступ к этой возможности, а не хоронить ее внутри некой абстракции. Назначение абстракций заключается в том, чтобы скрывать нежелательные свойства, а не полезные свойства. Например,

предположим, что у аппаратуры есть специальный способ перемещения больших участков изображений по экрану (то есть в видеопамяти) на высокой скорости. В этом случае ввод нового системного вызова, предоставляющего доступ к этому механизму, будет оправдан, так как это лучше, чем читать данные из видеоОЗУ в обычную память, а затем писать эти данные обратно в видеоОЗУ. Новый вызов должен просто перемещать биты в видеопамяти. Если этот новый системный вызов будет быстрым, это позволит пользователям создавать более эффективные программы. Если системный вызов медленный, никто не будет им пользоваться.

При проектировании системы возникает также вопрос использования ориентированных на соединение вызовов или вызовов, не использующих соединений. Стандартные системные вызовы UNIX и Win32 для чтения файлов являются ориентированными на соединение. Сначала вы открываете файл, затем читаете его и, наконец, закрываете файл. Некоторые протоколы работы с удаленными файлами также являются ориентированными на соединение. Например, чтобы использовать протокол FTP, пользователь сначала регистрируется на удаленной машине, читает файлы, а затем выходит из системы.

С другой стороны, некоторые протоколы удаленного доступа к файлам не требуют соединений. Например, протокол NFS не требует соединений, как было показано в главе 10. Каждый вызов NFS является независимым, поэтому файлы не открываются до их чтения или записи. И, разумеется, файлы не нужно закрывать после чтения или записи. Всемирная паутина также не требует соединений: чтобы прочитать web-страницу, вы просто запрашиваете ее. Не требуется никаких предварительных настроек (TCP-соединение все-таки требуется, но оно представляет собой более низкий уровень протокола; протокол HTTP, используемый для доступа к самой web-странице, не требует соединений).

От того, какой из механизмов выбрать — ориентированный на соединение или не требующий соединений, — зависит, будет ли от механизма требоваться дополнительная работа (например, по открытию файлов), или же эта работа перекладывается на плечи использующей механизм прикладной программы. В последнем случае получается существенный выигрыш в эффективности, если к одному и тому же файлу программа обращается много раз. Для системы ввода-вывода на одной машине, где стоимость подготовки ввода-вывода (открытия файла) низка, вероятно, лучше использовать стандартный способ (сначала открыть, затем использовать). Для удаленных файловых систем возможно использование обоих вариантов.

Другой вопрос, возникающий при проектировании интерфейса системных вызовов, заключается в его открытости. Список системных вызовов, определяемых стандартом POSIX, легко найти. Эти системные вызовы поддерживаются всеми системами UNIX, как и небольшое количество других вызовов, но полный список всегда публикуется. Корпорация Microsoft, напротив, никогда не публиковала список системных вызовов Windows 2000. Вместо этого публикуются функции интерфейса Win32 API, а также вызовы других интерфейсов; эти списки содержат огромное количество библиотечных вызовов (более 13 000 в Windows 2000), но только малое их число является настоящими системными вызовами. Аргумент в пользу открытости системных вызовов заключается в том, что программистам становится известна цена использования функций. Функции, исполняемые в про-

странстве пользователя, выполняются быстрее, чем те, которые требуют переключения в режим ядра. Закрытость системных вызовов также имеет свои преимущества, заключающиеся в том, что в результате достигается гибкость в реализации библиотечных процедур. То есть разработчики операционной системы получают возможность изменять действительные системные вызовы, сохраняя при этом работоспособность прикладных программ.

Реализация

Мы обсудили пользовательский интерфейс и интерфейс системных вызовов. Теперь пора обсудить вопрос реализации операционной системы. В следующих восьми разделах мы изучим некоторые общие концептуальные вопросы, относящиеся к стратегиям реализации. После этого мы рассмотрим некоторые примеры низкоуровневой техники, которая часто бывает полезной.

Структура системы

Вероятно, первым решением, которое должны принять разработчики системы, является решение о том, какой должна быть структура системы. Основные варианты мы изучили в разделе «Структура операционной системы» главы 1, но также рассмотрим их здесь. Монолитное неструктурированное устройство на самом деле представляет собой неудачную идею, подходящую разве что крошечной операционной системе для, например, холодильника, но даже в этом случае ее использование спорно.

Многоуровневые системы

Разумный подход, установившийся с годами, заключается в создании многоуровневых систем. Система THE, разработанная Э. Дейкстрой (см. табл. 1.3), была первой многоуровневой системой. У операционных систем UNIX (см. рис. 10.2) и Windows 2000 (см. рис. 11.2) также есть многоуровневая структура, но уровни в них в большей степени представляют собой способ описания системы, чем фактический руководящий принцип, использованный при ее построении.

При создании новой системы разработчики должны сначала очень тщательно выбрать уровни и определить функциональность каждого уровня. Нижний уровень всегда должен пытаться скрыть самые неприятные особенности аппаратуры, как это делает уровень HAL на рис. 11.2. Вероятно, следующий уровень должен обрабатывать прерывания, заниматься переключением контекста и работать с блоком управления памятью MMU, так что выше этого уровня код оказывается в основном машинно-независимым. На еще более высоких уровнях все зависит от вкусов и предпочтений разработчиков. Один вариант заключается в том, чтобы уровень 3 управлял потоками, включая планирование и синхронизацию потоков (рис. 12.1). При этом, начиная с уровня 4, мы получаем правильные потоки, которые нормально планируются и синхронизируются при помощи стандартного механизма (например, мьютексов).

Уровень					
7	Обработчик системных вызовов				
6	Файловая система 1	...	Файловая система m		
5	Виртуальная память				
4	Драйвер 1	Драйвер 2	...	...	...	Драйвер n
3	Потоки, планирование потоков, синхронизация потоков				
2	Обработка прерываний, переключение контекста, MMU				
1	Сокрытие низкоуровневой аппаратуры				

Рис. 12.1. Одна из возможных структур современной многоуровневой операционной системы

На уровне 4 мы обнаружим драйверы устройств, каждый из которых работает как отдельный поток, со своим состоянием, счетчиком команд, регистрами и т. д., возможно (но не обязательно), в адресном пространстве ядра. Такое устройство может существенно упростить структуру ввода-вывода, потому что когда возникает прерывание, оно может быть преобразовано в системный вызов `unlock` на мьютексе и обращение к планировщику, чтобы (потенциально) запустить новый готовый поток, который был блокирован мьютексом. Этот подход используется в системе MINIX, но в операционных системах UNIX, Linux и Windows 2000 обработчики прерываний реализованы как независимая часть системы, а не потоки, которые сами могут управляться планировщиком, приостанавливаться и т. д. Поскольку основная сложность любой операционной системы заключается во вводе-выводе, заслуживает внимания любой способ сделать его более удобным для обработки и более инкапсулированным.

Над уровнем 4, скорее всего, мы обнаружим виртуальную память, одну или несколько файловых систем и обработчики системных вызовов. Если виртуальная память расположена на более низком уровне, чем файловые системы, тогда блочный кэш может быть выгружаемым, что позволяет менеджеру виртуальной памяти динамически определять, как следует распределять физическую память между страницами пользователей и страницами ядра, включая кэш. Подобным образом работает операционная система Windows 2000.

Экзоядра

Хотя у многоуровневых структур есть много сторонников среди разработчиков систем, существует также другой лагерь, придерживающийся точно противоположной точки зрения [111]. Их точка зрения базируется на **сквозном аргументе** [286]. Эта концепция заключается в том, что если что-либо должно быть выполнено самой пользовательской программой, то неэффективно выполнять это также и на нижнем уровне.

Рассмотрим применение этого принципа к удаленному доступу к файлам. Если система заботится о том, чтобы файлы не были повреждены, она должна считать для каждого записываемого файла контрольную сумму и хранить ее вместе с файлом. Когда файл пересылается по сети, вместе с файлом пересылается также его контрольная сумма, которая проверяется по получении файла принимающей стороной. Если контрольные суммы не совпадают, файл пересылается снова.

Проверка контрольной суммы является более аккуратным методом, чем использование надежной сети. Она позволяет, помимо ошибок, возникающих при передаче файла по сети, перехватывать также ошибки чтения или записи на диск, ошибки памяти, программные ошибки в маршрутизаторах и т. д. Сквозной аргумент утверждает, что при этом использование надежной сети перестает быть необходимым, так как в конечной точке (у получающего процесса) есть достаточно информации, чтобы проверить правильность полученного файла самостоятельно. Единственный довод в пользу использования в данном случае надежного сетевого протокола заключается в повышении эффективности обработки сетевых ошибок на более низком уровне.

Сквозной аргумент может быть расширен почти до пределов всей операционной системы. Он утверждает, что операционная система не должна делать того, что пользовательская программа может сделать сама. Например, зачем нужна файловая система? Почему бы не позволить пользователю просто читать и писать участки диска защищенным способом? Конечно, большинству пользователей нравится работать с файлами, но, согласно сквозному аргументу, файловая система должна быть библиотечной процедурой, которую можно скомпоновать с любой программой, нуждающейся в файлах. Этот подход позволяет различным программам использовать различные файловые системы. Такая цепь рассуждений приводит к выводу, что операционная система должна только обеспечивать безопасное распределение ресурсов (например, процессорного времени или дисков) среди соревнующихся за них пользователей. Экзоядро представляет собой операционную систему, построенную в соответствии со сквозным аргументом [111].

Системы клиент-сервер

Компромиссом между операционной системой, которая делает все, и операционной системой, не делающей ничего, является операционная система, делающая кое-что. Результатом такого дизайна является схема микроядра, при которой большая часть операционной системы представляет собой серверные процессы, работающие на уровне пользователя (см. рис. 1.23). Такое устройство обладает наибольшей модульностью и гибкостью по сравнению с другими схемами. Предел гибкости заключается в том, чтобы каждый драйвер устройства также работал в виде пользовательского процесса, полностью защищенного от ядра и других драйверов. Удаление драйверов из ядра позволяет устранить наибольший источник нестабильности в любой операционной системе — полные ошибок драйверы, написанные третьими фирмами.

Разумеется, драйверы устройств должны получать доступ к аппаратным регистрам устройств, поэтому необходим определенный механизм, обеспечивающий этот доступ. Если аппаратура это позволяет, то каждому драйверному процессу может быть предоставлен доступ только к нужным ему устройствам ввода-вывода. Например, если регистры устройств ввода-вывода отображаются на адресное пространство памяти, то у каждого драйверного процесса может быть страница памяти, на которую отображаются регистры соответствующего устройства, но не другие страницы. Если же пространство портов ввода-вывода частично защищено, каждому драйверу может быть разрешен доступ к нужной ему порции этого пространства.

Даже если аппаратная поддержка недоступна, этой идеей все равно можно воспользоваться. Для этого нужен новый системный вызов, доступный только для драйверных процессов. Для работы этот системный вызов пользуется списком пар (порт, значение). Ядро сначала проверяет, владеет ли процесс всеми портами в списке. Если да, то оно копирует в порты соответствующие значения для инициализации устройства ввода-вывода. Подобный системный вызов может быть использован для чтения портов ввода-вывода защищенным образом.

Такой метод защищает структуры данных ядра от изучения и повреждения их драйверами устройств, что (по большей части) хорошо. Возможно создание аналогичного набора вызовов, позволяющим драйверам устройств читать и писать таблицы ядра, но только под контролем ядра и с его одобрения.

Главная проблема такого подхода, и проблема микроядер вообще, заключается в снижении производительности, вызываемом дополнительными переключениями контекста. Однако практически вся работа по созданию микроядер была выполнена много лет назад, когда центральные процессоры были значительно медленнее. Сегодня не так уж много приложений, использующих каждую каплю мощности процессора, которые не могут смириться с малейшей потерей производительности. В конце концов, когда работает текстовый редактор или web-браузер, центральный процессор простаивает около 90 % времени. Если операционная система, основанная на микроядре, превращает систему с процессором, работающем на частоте 900 МГц, в надежную систему, аналогичную по производительности системе с частотой 800 МГц, мало кто из пользователей станет жаловаться. Большинство пользователей были просто счастливы всего несколько лет назад, когда приобрели свой предыдущий компьютер с потрясающей тогда частотой процессора в 100 МГц.

Расширяемые системы

В обсуждавшихся выше системах клиент-сервер идея заключалась в том, чтобы вынести за пределы ядра столько, сколько возможно. Противоположный подход заключается в том, чтобы поместить больше модулей в ядро, но защищенным способом. Ключевое слово здесь, разумеется, «защищенным». Некоторые механизмы защиты, изучавшиеся в разделе «Мобильные программы» главы 9, предназначались для импортирования апплетов по Интернету, но они в той же мере применимы для установки чужеродного кода в ядро. Самыми важными из них являются «песочницы» и программы с электронной подписью, так как интерпретацию применять в ядре непрактично.

Конечно, сама расширяемая система не является методом структурирования операционной системы. Однако, начав с минимальной системы, состоящей в основном из механизма защиты, и постепенно добавляя к ядру защищенные модули, пока не будет достигнута требуемая функциональность, можно создать минимальную систему для конкретного приложения. При таком подходе новая операционная система может выкраиваться под каждое новое приложение благодаря включению только тех элементов, которые необходимы для данного приложения. Примером такой системы является Paramecium [336].

Потоки ядра

Еще один вопрос, имеющий отношение к данной теме, независимо от выбора структурной модели — это системные потоки. Иногда бывает удобно позволить существовать потокам ядра отдельно от пользовательских процессов. Эти потоки могут работать в фоновом режиме, записывая «грязные» страницы на диск, занимаясь свопингом процессов и т. д. На самом деле ядро само может целиком состоять из таких потоков. Когда пользователь обращается к системному вызову, пользовательский поток не выполняется в режиме ядра, а блокируется и передает управление потоку ядра, который принимает управление для выполнения работы.

Помимо потоков ядра, работающих в фоновом режиме, большинство систем также запускают в фоновом режиме множество процессов-демонов. Хотя они и не являются частью операционной системы, они часто выполняют «системные» функции. Это может быть получение и отправка электронной почты, а также обслуживание различных запросов удаленных пользователей, как, например, FTP или web-страницы.

Механизм и политика

Еще один принцип, помогающий добиться архитектурной согласованности, наряду с принципами минимализма и структурированности заключается в отделении механизма от политики. Если поместить механизм в операционную систему, а политику оставить пользовательским процессам, система может остаться неизменной, даже если появляется потребность в изменении политики. Даже если модуль, занимающийся политикой, должен располагаться в ядре, он должен быть, по возможности, изолирован от механизма, чтобы изменения в модуле политики не влияли на модуль механизма.

Чтобы сделать границу между механизмом и политикой отчетливей, рассмотрим два примера из реального мира. В качестве первого примера возьмем большую компанию, у которой есть отдел заработной платы, ответственный за выплату жалования сотрудникам. У него есть компьютеры, программное обеспечение, бланки, договоренность с банками, а также другие механизмы, требуемые для фактической выплаты зарплаты. Однако политика — принятие решений, кто и сколько получит — полностью отделена и является прерогативой управления. Отдел заработной платы просто выполняет порученную ему работу.

В качестве второго примера рассмотрим ресторан. Он обладает механизмом для обслуживания посетителей, включая столы, посуду, официантов, полную оборудования кухню, договоры с компаниями, продающими товары по кредитным карточкам и т. д. Политика, то есть что будет в меню, определяется шеф-поваром. Если шеф-повар решит, что тофу кончился, а большие бифштексы остались, то новая политика может быть выполнена существующим механизмом.

Теперь рассмотрим некоторые примеры из области операционных систем. Во-первых, взглянем на планирование потоков. В ядре может располагаться приоритетный планировщик с *k* уровнями приоритетов. Этот механизм представляет собой массив, проиндексированный уровнем приоритета, как показано на рис. 10.4 или на рис. 11.8. Каждый элемент массива представляет собой заголовок списка

готовых потоков с данным уровнем приоритета. Планировщик просто просматривает массив, начиная с максимального приоритета, выбирая первый подходящий поток. Политика заключается в установке приоритетов. В системе могут быть различные классы пользователей, например, у каждого пользователя может быть свой приоритет. Система может также позволять процессам устанавливать относительные приоритеты для своих потоков. Приоритеты могут увеличиваться после завершения операции ввода-вывода или уменьшаться, когда процесс истратил отпущенный ему квант. Может применяться также еще множество других политических подходов, но основной принцип состоит в разделении между установкой политики и претворением ее в жизнь.

Вторым примером является страничная подкачка. Механизм включает в себя управление блоком MMU, поддержку списка занятых и свободных страниц, а также программу для перемещения страниц с диска в память и обратно. Политика в данном случае заключается в принятии решения о выполняемых действиях при возникновении страничного прерывания. Политика может быть локальной или глобальной, основываться на алгоритме LRU, FIFO или каком-либо другом алгоритме, но она может (и должна) быть полностью отделена от механики фактической работы со страницами.

Третий пример — возможность загрузки модулей в ядро. Механизм определяет то, как они устанавливаются в ядро, как они связываются, к каким вызовам они могут обращаться и какие вызовы могут сами предоставлять. Политика определяет список пользователей, которым разрешается загружать модуль в ядро, а также список модулей, которые разрешается загружать каждому пользователю. Возможно, только суперпользователь может загружать модули, но разрешение загружать модули может также предоставляться и любому пользователю, если у модуля есть цифровая подпись соответствующей авторитетной организации.

Ортогональность

Хорошая системная организация включает в себя отдельные концепции, которые можно независимо смешивать. Например, в языке C есть примитивные типы данных, включающие целые, символьные числа и числа с плавающей точкой. Существуют механизмы для комбинирования типов данных, включая массивы, структуры и объединения. Эти концепции независимы друг от друга, что позволяет создавать целые массивы, символьные массивы, массивы структур, структуры, состоящие из массивов и т. д. Действительно, как только определен новый тип данных, например массив целых чисел, он может использоваться в качестве члена структуры или объединения, как если бы он был примитивным типом данных. Возможность независимо комбинировать раздельные концепции называется **ортогональностью**. Это свойство является прямым следствием простоты и полноты принципов.

Понятие ортогональности также встречается в операционных системах в различных формах. Одним из примеров является системный вызов `clone` операционной системы Linux, создающий новый поток. В качестве параметра этот вызов принимает битовый массив, задающий такие режимы, как независимое друг от друга совместное использование или копирование адресного пространства, рабочего каталога, дескрипторов файлов и сигналов. Если копируется все, то мы получаем

новый процесс, как и при использовании системного вызова fork. Если ничего не копируется, это означает, что создается новый поток в текущем процессе. Однако вероятно и создание промежуточных форм, невозможных в традиционных системах UNIX. Разделение свойств и их ортогональность позволяет получить более детальную степень управления.

Другое применение ортогональности — разделение понятий процесса и потока в Windows 2000. Процесс представляет собой контейнер для ресурсов, не более и не менее. Поток представляет собой объект планирования. Когда один процесс получает дескриптор от другого процесса, не имеет значения, сколько потоков у него есть. Когда планировщик выбирает поток, не важно, какому процессу он принадлежит. Эти понятия ортогональны.

Наш последний пример ортогональности возьмем из операционной системы UNIX. Создание процесса происходит здесь в два этапа: при помощи системных вызовов fork и exec. Создание нового адресного пространства и заполнение его новым образом памяти в данном случае разделены, что позволяет выполнить определенные действия между этими этапами. В операционной системе Windows 2000 эти два этапа нельзя разделить, то есть концепции создания нового адресного пространства и его заполнения новым образом памяти не являются ортогональными в этой системе. Последовательность системных вызовов clone и exec в системе Linux является еще более ортогональной, так как в данном случае возможно еще более детальное управление этими действиями. Общее правило может быть сформулировано следующим образом: наличие небольшого количества ортогональных элементов, которые могут комбинироваться различными способами, позволяет создать небольшую простую и элегантную систему.

Именование

У большинства долгоживущих структур данных, используемых операционной системой, есть имя или идентификатор, по которому к ним можно обращаться. Среди очевидных примеров можно назвать регистрационные имена, имена файлов, устройств, идентификаторов процессов и т. д. Способ создания и использования этих имен представляет собой важный вопрос при проектировании и реализации системы.

Имена, создаваемые для использования их людьми, представляют собой символьные строки формата ASCII или Unicode и, как правило, являются иерархическими. Так, иерархия отчетливо видна на примере путей файлов, например, путь */usr/ast/books/mos2/chap-12* состоит из имен каталогов, поиск которых следует начинать в корневом каталоге. Адреса URL также являются иерархическими. Например, URL www.cs.vu.nl/~ast/ указывает на определенную машину (*www*) определенного факультета (*cs*) определенного университета (*vu*) определенной страны (*nl*). Участок URL после косой черты обозначает определенный файл на указанной машине, в данном случае по умолчанию это файл *www/index.html* в домашнем каталоге пользователя ast. Обратите внимание, что URL (а также адреса DNS вообще, включая адреса электронной почты) пишутся «задом наперед», начинаясь с нижнего уровня дерева, в отличие от имен файлов, начинающихся с вершины дерева. На это можно взглянуть и по-другому, если положить дерево горизонталь-

но. При этом в одном случае дерево будет начинаться слева и расти направо, а в другом случае, наоборот, будет начинаться справа и расти влево.

Часто используется двухуровневое именование: внешнее и внутреннее. Например, к файлам всегда можно обратиться по имени, представляющему собой символьную строку. Кроме этого, почти всегда существует внутреннее имя, используемое системой. В операционной системе UNIX реальным именем файла является номер его i-узла. Имя в формате ASCII вообще не используется внутри системы. В действительности это имя даже не является уникальным, так как на файл может указывать несколько ссылок. А в операционной системе Windows 2000 в качестве внутреннего имени используется индекс файла в таблице MFT. Работа каталога заключается в преобразовании внешних имен во внутренние, как показано на рис. 12.2.

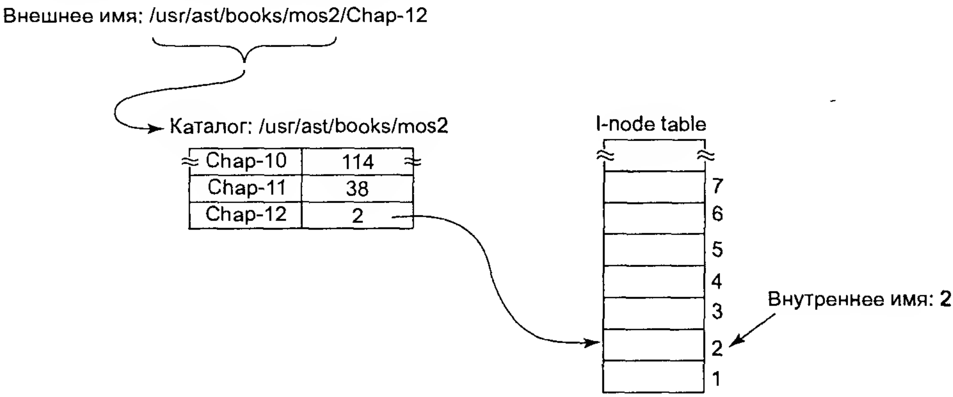


Рис. 12.2. Каталоги используются для преобразования внешних имен во внутренние

Во многих случаях (таких как приведенный выше пример с именем файла) внутреннее имя файла представляет собой уникальное целое число, служащее индексом в таблице ядра. Другие примеры имен-индексов: дескрипторы файлов в системе UNIX и дескрипторы объектов в Windows 2000. Обратите внимание, что ни у одного из данных примеров имен нет внешнего представления. Они предназначены исключительно для внутреннего использования системой и работающими процессами. В целом использование для временных имен табличных индексов, не сохраняющихся после перезагрузки системы, является удачным замыслом.

Операционные системы часто поддерживают несколько пространств имен, как внешних, так и внутренних. Например, в главе 11 мы рассмотрели три внешних пространства имен, поддерживаемых операционной системой Windows 2000: имена файлов, имена объектов и регистрационные имена (существует также пространство имен Active Directory, которое мы не обсуждали). Кроме того, существует бесчисленное количество внутренних пространств имен, для которых используются целые числа без знака — дескрипторы объектов, записи таблицы MFT и т. д. Хотя имена во внешних пространствах имен представляют собой строки формата Unicode, поиск файла по имени в реестре не будет работать, как не будет работать и попытка использования индекса MFT в таблице объектов. В правильно спроектированной операционной системе обязательно уделяется определенное внимание тому, сколько требуется пространств имен, какой синтаксис используется в каждом

из них, как можно отличить одно от другого, существуют ли абсолютные и относительные имена и т. д.

Время связывания

Как мы видели, для обращения к объектам в операционных системах используются различные типы имен. Иногда преобразование имени в объект является фиксированным, а иногда нет. В последнем случае может быть важным, когда имя связывается с объектом. Вообще говоря, **раннее связывание** проще, но не обладает гибкостью, тогда как **позднее связывание** сложнее, зато часто является более гибким.

Чтобы внести ясность в понятие времени связывания, давайте рассмотрим некоторые примеры из реального мира. Примером раннего связывания может быть практика некоторых колледжей, позволяющих родителям зачислять своего ребенка в колледж с момента рождения и вносить плату за его обучение заранее. Когда спустя 18 лет студент приходит в колледж, его обучение уже полностью оплачено, независимо от стоимости обучения на данный момент.

В промышленности ранним связыванием является заказ деталей заранее. Напротив, производство продукта точно в срок требует от поставщиков способности предоставлять требуемые детали прямо на месте, без предварительного заказа. Такая схема представляет собой пример позднего связывания.

Языками программирования часто поддерживаются различные виды связывания для переменных. Глобальные переменные связывает с конкретным виртуальным адресом компилятор. Это пример раннего связывания. Локальным переменным процедуры виртуальные адреса назначаются (в стеке) во время выполнения процедуры. Это пример промежуточного связывания. Переменным, хранящимся в куче (память для которых выделяется при помощи процедуры *malloc* в программах на языке C или процедуры *new* в программах на языке Java), виртуальный адрес назначается только на время их фактического использования. Это пример позднего связывания.

Для большей части структур данных в операционных системах чаще применяется раннее связывание, но иногда для гибкости используется позднее связывание. К этому вопросу имеет отношение выделение памяти. В ранних многозадачных системах из-за отсутствия аппаратной перенастройки адресов программу приходилось загружать по определенному адресу памяти и настраивать ее на работу по этому адресу. Если эта программа когда-либо выгружалась на диск, ее нужно было потом загрузить в те же адреса памяти, в противном случае она не могла работать. Напротив, виртуальная память с постраничной подкачкой представляет собой пример позднего связывания. Фактический физический адрес, соответствующий данному виртуальному адресу, неизвестен до тех пор, пока страница не будет физически перенесена в память.

Другим примером позднего связывания является размещение окна в графическом интерфейсе пользователя. В отличие от старых графических систем, в которых программист должен был указывать абсолютные экранные координаты для всех изображений, в современных графических интерфейсах пользователя программное обеспечение использует координаты относительно исходного окна. Такие координаты неизвестны до тех пор, пока это окно не будет выведено на экран, и даже могут быть со временем изменены.

Статические и динамические структуры

Разработчики операционных систем постоянно вынуждены выбирать между статическими и динамическими структурами данных. Статические структуры всегда проще для понимания, легче для программирования и быстрее в использовании. Динамические структуры обладают большей гибкостью. Очевидный пример — таблица процессов. В ранних системах для каждого процесса просто выделялся фиксированный массив структур. Если таблица процесса состояла из 256 элементов, тогда в любой момент могло существовать не более 256 процессов. Попытка создания 257-го процесса заканчивалась неудачей по причине отсутствия места в таблице. Аналогичные соображения были справедливы для таблицы открытых файлов (как для каждого пользователя в отдельности, так и для системы в целом) и различных других таблиц ядра.

Альтернативная стратегия заключается в создании таблицы процессов в виде связного списка мини-таблиц, начиная с одной-единственной мини-таблицы. Когда эта таблица заполняется, в глобальном пуле выделяется память под следующую таблицу, которая связывается с первой таблицей. Таким образом, таблица процесса не переполнится, пока не закончится вся память у ядра.

С другой стороны, использование связных списков приводит к усложнению программы поиска записей в таблице. Например, программа для поиска идентификатора процесса *pid* в статической таблице процессов показана в листинге 12.2. Эта программа проста и эффективна. Чтобы выполнить ту же задачу для связного списка мини-таблиц, потребуется больше работы.

Листинг 12.2. Программа для поиска в таблице идентификатора процесса

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
      if (p->proc_pid == pid) {
         found = 1;
         break;
      }
}
```

Статические таблицы лучше всего использовать, когда имеется много памяти или когда заранее можно довольно точно предсказать размер таблицы. Например, в однопользовательской системе маловероятно, что пользователь запустит одновременно более 32 процессов, и поэтому не случится катастрофы, если пользователю будет отказано в запуске 33-го процесса.

Альтернативный метод заключается в использовании таблицы фиксированного размера, но с выделением, когда эта таблица заполнится, новой таблицы фиксированного размера, например, в два раза большей. При этом текущие записи копируются из старой таблицы в новую, а память, которая была занята старой таблицей, возвращается в пул. Таким образом, таблица всегда остается непрерывной, а не связной. Недостаток этого подхода состоит в том, что требуется определенное управление памятью, кроме того, адрес таблицы будет переменным вместо константы.

То же самое относится к стекам ядра. Когда поток переключается в режим ядра или когда запускается поток в режиме ядра, этому потоку требуется стек в адрес-

ном пространстве ядра. Для потоков пользователя стек можно инициализировать так, чтобы он рос вниз от вершины виртуального адресного пространства. При этом его размер можно заранее не указывать. Для потоков ядра размер стека должен быть указан заранее, так как стек занимает часть виртуального адресного пространства ядра, кроме того, стеков может быть несколько. Вопрос заключается в следующем: сколько памяти следует выделить для каждого стека? Преимущества и недостатки различных подходов здесь примерно такие же, как и в случае с таблицей процессов.

Проблема выбора между статическим или динамическим выделением памяти возникает также для планирования процессов. В некоторых системах, особенно системах реального времени, планирование может быть выполнено заранее статически. Например, авиалинии знают, в котором часу их самолеты будут взлетать, за несколько недель до их фактического отправления. Подобно этому, мультимедийным системам известно заранее, когда запускать те или иные видео-, аудио- и другие процессы. Для универсальных систем общего назначения эти соображения не работают, и планирование должно быть динамическим.

Вопрос выбора между статическим или динамическим выделением памяти возникает и при проектировании структур ядра. Значительно проще, если ядро построено, как единая двоичная программа, загружаемая в память для работы. Следствием такого подхода, однако, является то, что для установки каждого нового устройства ввода-вывода необходима перекомпоновка ядра вместе с драйвером нового устройства. Подобным образом работали ранние версии операционной системы UNIX, что всех устраивало, так как новые устройства ввода-вывода добавлялись к мини-компьютерам довольно редко. Сегодня большинство операционных систем позволяют динамически добавлять программы в ядро, со всеми дополнительными сложностями, которые такой подход влечет за собой.

Реализация системы сверху вниз и снизу вверх

Хотя лучше всего проектировать систему сверху вниз, теоретически реализация системы может выполняться как сверху вниз, так и снизу вверх. При реализации сверху вниз конструкторы начинают с обработчиков системных вызовов, а затем смотрят, какие механизмы и структуры данных требуются для их поддержки. Затем пишутся эти процедуры, при этом весь процесс повторяется до тех пор, пока не будет достигнут аппаратный уровень.

Недостаток этого подхода заключается в том, что, пока в наличии имеются только процедуры верхнего уровня, трудно что-либо протестировать. По этой причине многие разработчики считают более практичным построение системы снизу вверх. При таком подходе сначала пишется программа, скрывающая аппаратуру нижнего уровня (уровень HAL на рис. 11.2). Обработка прерываний и драйвер часов также оказываются нужны уже на раннем этапе конструирования.

Затем можно реализовать многозадачность вместе с простым планировщиком (например, запускающим процессы в порядке циклической очереди). Уже в этот момент система может быть протестирована, чтобы проверить, правильно ли она управляет несколькими процессами. Если все работает нормально, можно присту-

пить к детальной разработке различных таблиц и структур данных, необходимых системе, особенно тех, которые управляют процессами и потоками, а также памятью. Ввод-вывод и файловую систему можно отложить на потом, реализовав поначалу лишь примитивный ввод с клавиатуры и вывод на экран для тестирования и отладки. В некоторых случаях следует защитить ключевые низкоуровневые структуры данных, разрешив доступ к ним только с помощью специальных процедур доступа — в результате мы получаем объектно-ориентированное программирование, независимо от того, какой язык программирования применяется в действительности. Когда нижние уровни созданы, они могут быть тщательно протестированы. Таким образом, система создается снизу вверх, подобно тому, как строятся высокие здания.

Если над проектом работает большая команда программистов, тогда альтернативный подход заключается в том, чтобы сначала создать детальный проект всей системы, после чего распределить задачи по написанию отдельных модулей среди различных групп программистов. Каждая группа независимо тестирует собственную работу. Когда все модули готовы, они объединяются и тестируются совместно. Недостаток такого подхода заключается в том, что если изначально ничто не работает, очень трудно определить, который модуль создан правильно, а какой содержит ошибки и какая группа программистов неверно поняла, чего ей следует ожидать от других модулей. Тем не менее такой метод часто применяется при наличии большого количества программистов, чтобы максимально использовать распараллеливание работ по конструированию системы.

Полезные методы

Итак, мы только что обсудили некоторые абстрактные идеи, применяющиеся при проектировании и конструировании операционных систем. Теперь мы рассмотрим несколько конкретных методов, полезных при реализации систем. Разумеется, существует также множество других методов, но объем книги не позволяет нам рассмотреть все методы.

Сокрытие аппаратуры

Большое количество аппаратуры весьма неудобно в использовании. Его следует скрывать на ранней стадии реализации системы (если только оно не предоставляет особых возможностей). Некоторые детали самого нижнего уровня могут быть скрыты уровнем вроде HAL, показанным на рис. 12.1. Однако есть детали, которые таким способом скрыть невозможно.

На ранней стадии конструирования системы следует решить вопрос обработки прерываний. Наличие прерываний делает программирование неприятным делом, но операционные системы должны работать с прерываниями. Один из подходов состоит в том, чтобы немедленно преобразовать их во что-либо иное. Например, каждое прерывание можно превратить в появляющийся новый поток. При этом более высокие уровни будут иметь дело уже не с прерываниями, а с потоками.

Второй метод заключается в том, чтобы преобразовать каждое прерывание в операцию unlock на мьютексе, которого ожидает соответствующий драйвер. Тогда единственный эффект от прерывания будет заключаться в том, что один из потоков перейдет в состояние готовности.

Третий подход состоит в преобразовании прерывания в сообщение какому-либо потоку. Программа низкого уровня просто формирует сообщение, в котором содержатся сведения о том, откуда прибыло прерывание, ставит его в очередь и вызывает планировщик, который, возможно, запустит процесс, вероятно, ожидающий этого сообщения. Все эти методы, а также подобные им другие, пытаются преобразовать прерывания в операции синхронизации потоков. Обработкой каждого прерывания соответствующим потоком в соответствующем контексте проще управлять, чем создать обработчик прерываний, работающий в произвольном контексте, то есть в том, в котором случилось прерывание. Разумеется, обработка прерываний должна быть эффективной, но глубоко внутри операционной системы эффективным должно быть все.

Большинство операционных систем спроектировано так, чтобы работать на различных платформах. Эти платформы могут отличаться центральными процессорами, блоками MMU, длиной машинного слова, объемом оперативной памяти и другими параметрами, которые трудно замаскировать уровнем HAL или его эквивалентом. Тем не менее желательно иметь единый набор исходных файлов, используемых для формирования всех версий. В противном случае каждую обнаруженную ошибку придется исправлять много раз во многих исходных файлах. При этом возникает опасность, что исходные файлы будут отличаться друг от друга все больше и больше.

С некоторыми различиями аппаратуры, такими как объем ОЗУ, можно бороться, оформив этот параметр в виде переменной и определяя его значение во время загрузки системы. Например, переменная, содержащая объем оперативной памяти, может использоваться программой, предоставляющей память процессам, а также для определения размера блока кэша, таблиц страниц и т. д. Даже размер статических таблиц, таких как таблица процессов, может задаваться при загрузке, в зависимости от общего объема оперативной памяти.

Однако другие различия, такие как различия в центральных процессорах, не могут быть скрыты при помощи единого двоичного кода, определяющего при загрузке тип процессора. Один из способов решения данной проблемы состоит в использовании условной трансляции единого исходного кода. В исходных файлах для различных конфигураций используются определенные флаги компиляции, позволяющие получать из единого исходного текста различные двоичные файлы в зависимости от центрального процессора, длины слова, блока MMU и т. д. Представьте себе операционную систему, которая должна работать на компьютерах с процессорами Pentium и UltraSPARC, требующих различных программ инициализации. Процедура *init* может быть написана так, как показано в листинге 12.3, *а*. В зависимости от значения константы *CPU*, определяемой в заголовочном файле *config.h*, будет выполняться либо один тип инициализации, либо другой. Поскольку в двоичном файле содержится только один вариант программы, потери эффективности не происходит.

Листинг 12.3. Условная компиляция, зависящая от типа центрального процессора (*а*); условная компиляция, зависящая от длины слова (*б*)

```
#include "config.h"
init( )
{
#if (CPU == PENTIUM)
/* Здесь инициализация для Pentium */
#endif

#if (CPU == ULTRASPARC)
/* Здесь инициализация для UltraSPARC */
#endif
}
```
а

```
#include "config.h"
#if (WORD_LENGTH == 32)
typedef int Register;
   #endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```
б

В качестве второго примера предположим, что нам требуется тип данных *Register*, который должен состоять из 32 бит на компьютере с процессором Pentium и 64 бит на компьютере с процессором UltraSPARC. Этого можно добиться при помощи условного кода в листинге 12.3, *б* (при условии, что компилятор воспринимает тип int, как 32-разрядное целое, а тип long — как 64-разрядное). Как только это определение дано (возможно, в заголовочном файле, включаемом во все остальные исходные файлы), программист может просто объявить переменные типа *Register* и быть уверенным, что эти переменные имеют правильный размер.

Разумеется, заголовочный файл *config.h* должен быть определен корректно. Для процессора Pentium он может выглядеть примерно так:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

Чтобы откомпилировать операционную систему для процессора UltraSPARC, нужно использовать другой файл *config.h*, содержащий правильные значения для процессора UltraSPARC, возможно, что-то вроде

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Некоторые читатели могут удивиться, почему переменные *CPU* и *WORD_LENGTH* управляются различными макросами. В определении константы *Register* можно сделать ветвление программы, устанавливая ее значение в зависимости от значения константы *CPU*, то есть устанавливая значение константы *Register* равной 32 бит для процессора Pentium и 64 бит для процессора UltraSPARC. Однако эта идея не слишком удачна. Что произойдет, если позднее мы соберемся переносить систему на 64-разрядный процессор Intel Itanium? Для этого нам пришлось бы добавить третий условный оператор в листинг 12.3, *б* для процессора Itanium. При том, как это было сделано, нужно только добавить строку

```
#define WORD_LENGTH 64
```

в файл *config.h* для процессора Itanium.

Этот пример иллюстрирует обсуждавшийся ранее принцип ортогональности. Участки системы, зависящие от типа центрального процессора, должны компилироваться с использованием условной компиляции, в зависимости от значения константы *CPU*, а для участков системы, зависимых от размера слова, должен исполь-

зоваться макрос *WORD_LENGTH*. Те же соображения справедливы и для многих других параметров.

Косвенность

Иногда говорят, что нет такой проблемы в кибернетике, которую нельзя решить на другом уровне косвенности. Хотя это определенное преувеличение, во фразе имеется и доля истины. Рассмотрим несколько примеров. В системах на основе процессора Pentium при нажатии клавиши аппаратура формирует прерывание и помещает в регистр устройства не символ ASCII, а скан-код клавиши. Более того, когда позднее клавиша отпускается, генерируется второе прерывание, также с номером клавиши. Такая косвенность предоставляет операционной системе возможность использовать номер клавиши в качестве индекса в таблице, чтобы получить по его значению символ ASCII. Этот способ облегчает обработку разных клавиатур, существующих в различных странах. Наличие информации как о нажатии, так и об отпускании клавиш позволяет использовать любую клавишу в качестве регистра, так как операционной системе известно, в котором порядке нажимались и отпускались клавиши.

Косвенность также используется при выводе данных. Программы могут выводить на экран символы ASCII, но эти символы могут интерпретироваться как индексы в таблице, содержащей текущий отображаемый шрифт. Элемент таблицы содержит растровое изображение символа. Такая форма косвенности позволяет отделить символы от шрифта.

Еще одним примером косвенности служит использование старших номеров устройств в UNIX. В ядре содержатся две таблицы, одна для блочных устройств и одна для символьных, индексированные старшим номером устройства. Когда процесс открывает специальный файл, например */dev/hd0*, система извлекает из i-узла информацию о типе устройства (блочное или символьное), а также старший и младший номера устройств и, используя их в качестве индексов, находит в таблице драйверов соответствующий драйвер. Такой вид косвенности облегчает реконфигурацию системы, так как программы имеют дело с символьными именами устройств, а не с фактическими именами драйверов.

Еще один пример косвенности встречается в системах передачи сообщений, указывающих в качестве адресата не процесс, а почтовый ящик. Таким образом достигается существенная гибкость (например, секретарша может принимать почту своего шефа).

В определенном смысле использование макросов, как, например,

```
#define PROC_TABLE_SIZE 256
```

также представляет собой одну из форм косвенности, так как программист может написать программу, не зная фактической величины таблицы. Считается хорошей практикой давать символьные имена всем константам (иногда кроме –1, 0 и 1) и помещать их в заголовки с соответствующими комментариями.

Повторное использование

Часто возникает возможность использовать повторно ту же самую программу в несколько отличном контексте. Это позволяет уменьшить размер двоичных

файлов, а кроме того означает, что такую программу потребуется отлаживать всего один раз. Например, предположим, что для учета свободных блоков на диске используются битовые массивы. Дисковыми блоками можно управлять при помощи процедур *alloc* и *free*.

Как минимум эти процедуры должны работать с любым диском. Но мы можем пойти дальше в этих рассуждениях. Те же самые процедуры могут применяться для управления блоками памяти, блоками кэша файловой системы и i-узлами. В самом деле, они могут использоваться для распределения и освобождения ресурсов, которые могут быть линейно пронумерованы.

Реентерабельность

Реентерабельность означает свойство программы, позволяющее одновременно выполнять эту программу нескольким процессами. На мультипроцессорах всегда имеется опасность, что один из процессоров начнет выполнение процедуры, уже выполняющейся другим процессором. В этом случае два (или более) потока на различных центральных процессорах будут одновременно выполнять одну и ту же программу. Если в этой программе существуют области, для которых такая ситуация нежелательна, доступ к этим (критическим) областям должен защищаться при помощи мьютексов.

Однако в однопроцессорных системах эта проблема также существует. В частности, большая часть операционных систем работает с разрешенными прерываниями. Если прерывания запрещать, то многие сигналы, подаваемые устройствами ввода-вывода, будут потеряны и система станет ненадежной. В то время когда операционная система выполняет некоторую процедуру, может произойти прерывание, и вполне возможно, что обработчик прерываний также начнет выполнение этой же процедуры. Если на момент прерывания структуры данных в прерванной процедуре находились в противоречивом состоянии (то есть прерванная процедура начала изменять эти данные, но не успела закончить), обработчик прерываний либо будет работать некорректно, либо не сможет работать вообще.

Такая ситуация может, например, произойти в том случае, если прерываемой процедурой является сам планировщик. Предположим, что некий процесс использовал свой квант, и операционная система переместила его в конец очереди. Во время работы со списком происходит прерывание, в результате которого другой процесс переходит в состояние готовности и запускает планировщик. Если в этот момент очередь будет находиться в противоречивом состоянии, операционная система, скорее всего, не сможет продолжать работу. Поэтому даже в однопроцессорной системе лучше всего большую часть системы делать реентерабельной, критические структуры данных защищать мьютексами, а прерывания в некоторых случаях вообще запрещать.

Метод грубой силы

Применение простых решений под названием метода грубой силы с годами приобрело негативный оттенок, однако простота решения часто оказывается преимуществом. В каждой операционной системе есть множество процедур, которые редко вызываются или которые оперируют таким небольшим количеством данных, что оптимизировать их нет смысла. Например, в системе часто приходится искать какой-

либо элемент в таблице или массиве. Метод грубой силы в данном случае заключается в том, чтобы оставить таблицу в том виде, в каком она есть, никак не упорядочивая элементы, и производить поиск в ней линейно от начала к концу. Если число элементов в таблице невелико (например, не более 100), выигрыш от сортировки таблицы или применения хэширования будет невелик, но программа станет гораздо сложнее и, следовательно, вероятность содержания в ней ошибок резко возрастет.

Разумеется, для функций, находящихся в критических участках системы, например в процедуре, занимающейся переключением контекста, следует предпринять все меры для их ускорения, возможно даже писать их (Боже упаси!) на ассемблере. Но большая часть системы не находится в критическом участке. Так, ко многим системным вызовам редко обращаются. Если системный вызов fork выполняется раз в 10 с, а его выполнение занимает 10 мс, тогда, даже если удастся невозможное — оптимизация, после которой выполнение системного вызова fork будет занимать 0 мс, — общий выигрыш составит всего 0,1 %. Если после оптимизации код станет больше и будет содержать больше ошибок, то в данном случае лучше оптимизацией не заниматься.

Проверка на ошибки прежде всего

Многие системные вызовы могут завершиться безуспешно по многим причинам: файл, который нужно открыть, может принадлежать кому-либо другому; создание процесса может не удаться, так как таблица процессов переполнена; сигнал не может быть послан, потому что процесса-получателя не существует. Операционная система должна скрупулезно проверить возможность наличия самых разных ошибок, прежде чем выполнять системный вызов.

Для выполнения многих системных вызовов требуется получение ресурсов, например элементов таблицы процессов, элементов таблицы i-узлов или дескрипторов файлов. Прежде чем захватывать ресурсы, полезно проверить, можно ли выполнить этот системный вызов. Это означает, что всю проверку следует поместить в начало процедуры, выполняющей системный вызов. Каждая проверка должна иметь следующий вид:

```
if (error_condition) return(ERROR_CODE);
```

Если системному вызову удается пробраться сквозь все тесты, тогда становится ясно, что он завершится успешно. В этот момент ему можно выделять ресурсы.

Если проверки будут перемежаться обращением к ресурсам, то при неудачном результате очередного теста системному вызову придется возвращать все полученные ресурсы. Если программа написана не совсем корректно и какой-либо ресурс не будет возвращен, операционная система сразу не зависнет. Например, один из элементов таблицы процессов может оказаться навечно (до перезагрузки) недоступным. Однако со временем эта ситуация может повториться несколько раз, при этом количество недоступных ресурсов будет накапливаться. Наконец, большая часть элементов таблицы процессов может стать недоступной, что приведет к зависанию системы, причем исправление этой ошибки, скорее всего, окажется крайне сложным, так как воспроизвести эту ситуацию будет непросто.

Многие системы страдают подобными «заболеваниями» в форме утечки памяти. Довольно часто программы обращаются к процедуре *malloc*, чтобы получить

память, но забывают позднее обратиться к функции *free*, чтобы освободить ее. Вся память системы постепенно исчезает, пока система не зависает.

Энглер и его коллеги [109] предложили интересный метод проверки на наличие подобных ошибок во время компиляции. Авторы этой книги выяснили, что программистам известно множество условий, за соблюдением которых им приходится следить при написании программы и которые не проверяются компилятором. Так, например, если вы заблокировали мьютекс, то все пути выполнения этой программы, начиная с этого места, должны содержать разблокировку мьютекса и не должны содержать повторной блокировки того же мьютекса. Авторы книги разработали метод, позволяющий программисту поручить компилятору автоматически следить за соблюдением подобных правил. Программист также может указать в программе, что выделенная процессу память должна быть освобождена независимо от того, по какой ветви будет продолжаться выполнение программы, а также поручить компилятору следить за выполнением множества других условий.

Производительность

При прочих равных условиях быстрая операционная система лучше медленной. Однако быстрая, но ненадежная операционная система хуже надежной, но медленной. Поскольку сложные оптимизирующие методы часто приводят к появлению в системе новых ошибок, не следует злоупотреблять оптимизацией. И все же существуют области, в которых производительность является критичной и оптимизация стоит затрачиваемых усилий. В следующих разделах мы рассмотрим несколько общих методов, которые могут применяться для повышения производительности там, где это нужно.

Почему операционные системы такие медленные?

Прежде чем перейти к разговору о методах оптимизации, имеет смысл отметить, что в медленности работы многих операционных систем в достаточной мере виноваты сами операционные системы. Например, старые операционные системы, такие как MS-DOS и UNIX Version 7, загружались за несколько секунд. Для загрузки современных версий системы UNIX и системы Windows 2000 требуется несколько минут, несмотря на то, что они загружаются на аппаратуре, работающей в 100 раз быстрее. Причина состоит в том, что новые системы выполняют гораздо больше действий, нужно это или нет. Например, plug-and-play облегчает установку новых аппаратных устройств, но платой за это является то, что при *каждой* загрузке операционная система должна исследовать все аппаратное обеспечение, чтобы определить, не появилось ли что-либо новое. Сканирование шин занимает время.

Альтернативный (и, по мнению автора, лучший) подход заключается в том, чтобы совсем выбросить plug-and-play, а на экране установить значок, помеченный «Установка новой аппаратуры». Установив новое аппаратное устройство, пользо-

ватель просто щелкает мышью на этом значке, запуская процедуру сканирования шин, вместо того чтобы разрешать операционной системе делать это при каждой загрузке. Разработчики операционных систем, безусловно, знали о наличии такой возможности. Однако они отказались от нее, в основном потому, что считали пользователей не достаточно умными, чтобы правильно выполнить требуемые действия (правда, высказано это было в более мягких выражениях). Это всего лишь один пример, но можно привести и множество других примеров того, как желание сделать систему «дружественной по отношению к пользователю» (или «защищенной от дурака», в зависимости от вашей точки зрения) значительно снижало производительность системы.

Вероятно, больше всего могут добиться разработчики систем в деле увеличения производительности, если существенно повысят избирательность при добавлении к системе новых функций. Вопрос, который должен при этом ставиться, не «Понравится ли это пользователям?», а «Стоит ли добавление этой функции той неизбежной платы, заключающейся в увеличении программы, снижении скорости, увеличении сложности и снижении надежности?» Следует включать новую функцию, только если преимущества со всей очевидностью перевешивают недостатки. Некоторые программисты склонны предполагать, что размеры программы будут равны 0, а скорость ее работы будет бесконечной. Как показывают эксперименты, эта точка зрения является несколько оптимистичной.

Другой фактор, также играющий роль в данном вопросе, заключается в рыночной стратегии производителей программного обеспечения. К тому времени, когда версия 4 или 5 некоего программного продукта попадает на рынок, все нужные новые функции, включенные в эту версию, у потребителей, вероятно, уже будут. Чтобы не снижать уровня продаж, многие производители продолжают производить новые версии с большим количеством функций. Добавление новых функций просто ради добавления новых функций может помочь увеличению продаж, но редко способствует увеличению производительности.

Что следует оптимизировать?

Общее правило гласит, что первая версия системы должна быть как можно проще. Оптимизировать следует только те части системы, которые, очевидно, будут представлять собой проблему, поэтому их оптимизация является неизбежной. Одним из таких примеров является наличие блочного кэша для файловой системы. Как только операционная система отлажена до работоспособного состояния, следует произвести тщательные измерения, чтобы понять, на что *действительно* тратится время. Опираясь на эти числа, следует заниматься оптимизацией в тех областях, в которых это будет наиболее полезно.

Вот правдивая история о том, как оптимизация принесла больше вреда, чем пользы. Один из студентов автора (имени студента мы здесь называть не будем) написал программу *mkfs* для системы MINIX. Эта программа создает пустую файловую систему на только что отформатированном диске. На оптимизацию этой программы студент затратил около 6 месяцев. Когда он попытался запустить эту

программу, оказалось, что она не работает, после чего потребовалось еще 6 дополнительных месяцев на ее отладку. На жестком диске эту программу, как правило, запускают всего лишь один раз, при установке системы. Она также только раз запускается для каждого гибкого диска — после его форматирования. Каждый запуск программы занимает около 2 с. Даже если бы работа неоптимизированной версии занимала 1 мин, то затрата такого большого времени на оптимизацию столь редко используемой программы являлась бы непроизводительным расходованием ресурсов.

Лозунг, применимый к оптимизации производительности, мог бы звучать так:

Лучшее — враг хорошего.

Под этим мы подразумеваем, что как только удается достичь приемлемого уровня производительности, то попытки выжать последние несколько процентов, видимо, уже не стоят затрачиваемых усилий и усложнения программы. Если алгоритм планирования достаточно хорош и обеспечивает 90-прцентную загрузку центрального процессора, возможно, этого достаточно. Разработка значительно более сложного алгоритма, на 5 % лучше имеющегося, не всегда представляет собой удачную идею. Аналогично, если частота подкачки страниц достаточно низка, то есть подкачка не представляет собой узкое место, то, как правило, нет смысла лезть из кожи вон, чтобы добиться оптимальной производительности. Недопущение сбоев в работе системы представляется намного более важной задачей, нежели достижение оптимальной производительности, особенно если алгоритм, оптимальный при одном уровне загруженности компьютера, может оказаться неоптимальным при другом уровне.

Выбор между оптимизацией по скорости и по занимаемой памяти

При увеличении производительности программы приходится идти на компромисс между сокращением времени выполнения программы и увеличением занимаемой ею памяти. Программисты часто оказываются перед выбором между медленным, но компактным алгоритмом и более быстрым, но занимающим значительно больше памяти. Занимаясь важной оптимизацией, следует искать алгоритмы, дающие увеличение скорости за счет использования большего объема памяти или, наоборот, сберегающие драгоценную память за счет выполнения большего количества вычислений.

Часто может быть полезен метод, заключающийся в замене небольших процедур макросами. Использование макроса устраняет накладные расходы, связанные с вызовом процедуры. Выигрыш оказывается особенно существенным, если процедура вызывается многократно в цикле. Например, предположим, что мы используем битовый массив для учета ресурсов и нам часто приходится узнавать, сколько единиц свободно в некоторой области битового массива. Для этой цели нам нужна процедура *bit_count*, считающая количество единичных битов в байте. Простая процедура показана в листинге 12.4, *а*. Эта процедура в цикле перебирает биты в байте, считая их по одному на каждом цикле.

Листинг 12.4. Процедура для подсчета битов в байте (*а*); макрос для подсчета битов (*б*); макрос для поиска числа битов в таблице (*в*)

```
#define BYTE_SIZE 8                     /* Байт содержит 8 бит */
int bit_count(int byte)
{                                       /* Подсчет битов в байте */
int i. count = 0:
for (i = 0: i < BYTE_SIZE: i++)         /* Цикл по битам байта */
if ((byte >> i) & 1) count++:           /* Если бит равен 1. увеличить на единицу сумму */
return(count):                          /* Вернуть сумму */
}
```

а

```
/* Макрос для подсчета суммы битов в байте */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)
```

б

```
/* Макрос для поиска числа битов в таблице */
char bits[256] = {0. 1. 1. 2. 1. 2. 2. 3. 1. 2. 2. 3. 2. 3. 3. 4. 1. 2. 2. 3. 2. 3. 3. ...}:
#define bit_count(b) (int) bits[b]
```

в

У этой процедуры два источника неэффективности. Во-первых, ей нужно передавать управление, для чего требуется выделение стека, а после работы процедура должна вернуть результат и управление. Эти накладные расходы есть у каждого вызова процедуры. Во-вторых, процедура содержит цикл, а с циклом всегда связаны определенные накладные расходы.

Полностью отличный подход заключается в использовании макроса в листинге 12.4, *б*. Это выражение вычисляет сумму битов, последовательно сдвигая аргумент и выделяя при помощи маски младший бит. Этот макрос трудно назвать произведением искусства, но он встречается в программе всего один раз. Вызов этого макроса выглядит идентично вызову процедуры:

```
sum = bit_count(table[i]):
```

Таким образом, если не считать несколько беспорядочного определения, макрос выглядит не хуже процедуры, но он является намного более эффективным, так как в случае макроса устраняются как накладные расходы процедурного вызова, так и накладные расходы цикла.

Оптимизация данного алгоритма может быть продолжена. Зачем вообще считать сумму битов? Почему не посмотреть результат в таблице? В конце концов, байт может принимать всего 256 значений, а число бит в байте может быть от 0 до 8. Мы можем объявить массив *bits* из 256 значений, содержащий значения сумм битов, инициализируемые во время компиляции программы. При таком методе во время работы программы потребуется всего одна команда обращения к массиву по индексу. Соответствующий макрос показан в листинге 12.4, *в*.

Показанные выше макросы представляют собой понятные примеры выигрыша в скорости за счет увеличения размеров программы. Однако мы могли бы пойти в деле оптимизации еще дальше. Если требуется количество битов в 32-разрядном слове, то с помощью макроса *bit_count* нам потребовалось бы для каждого слова выполнить четыре обращения к массиву. Если мы увеличим размер табли-

цы до 65 536 элементов, мы можем обойтись всего двумя обращениями к таблице на слово за счет значительного увеличения таблицы.

Поиск значений в таблице может использоваться и в других ситуациях. Например, в главе 7 мы обсуждали работу алгоритма сжатия JPEG, в котором применялось довольно сложное дискретное косинусное преобразование. В альтернативном методе сжатия GIF для кодирования 24-разрядных пикселов формата RGB применяется поиск в таблице. Однако алгоритм GIF работает только с изображениями, содержащими не более 256 цветов. Для каждого сжимаемого изображения формируется палитра из 256 цветов, хранящихся в формате RGB. Сжатое изображение состоит из 8-разрядных индексов таблицы вместо 24-разрядных значений цвета, благодаря чему достигается сжатие в три раза. Эта идея проиллюстрирована для фрагмента изображения 4×4 пиксела на рис. 12.3. Несжатое изображение показано на рис. 12.3, *а*. Каждый пиксел здесь представляет собой 24-разрядное число, в котором каждый из трех байтов содержит интенсивность красного, зеленого и синего цвета. Соответствующее этому фрагменту изображение в формате GIF показано на рис. 12.3, *б*. Палитра цветов, показанная на рис. 12.3, *в*, хранится прямо в файле GIF. В действительности формат GIF несколько сложнее, но основная идея заключается в применении таблицы цветов.

24 бит

3,8,13	3,8,13	26,4,9	90,2,6
3,8,13	3,8,13	4,19,20	4,6,9
4,6,9	10,30,8	5,8,1	22,2,0
10,11,5	4,2,17	88,4,3	66,4,43

а

8 бит

7	7	2	6
7	7	3	4
4	5	10	0
8	9	2	11

б

24 бит

11	66,4,43
10	5,8,1
9	4,2,17
8	10,11,5
7	3,8,13
6	90,2,6
5	10,30,8
4	4,6,9
3	4,19 20
2	88,4,3
1	26,4,9
0	22,2,0

в

Рис. 12.3. Фрагмент несжатого изображения с 24 битами на пиксел (а); тот же фрагмент, сжатый при помощи алгоритма GIF с 8 битами на пиксел (*б*); палитра цветов (*в*)

Существует другой способ уменьшить размер изображения, служащий иллюстрацией еще одного компромисса. Для описания изображений может использоваться язык программирования PostScript. (В действительности для этой цели может применяться любой язык, но PostScript предназначен именно для этого.) Многие принтеры имеют встроенный интерпретатор языка PostScript.

Например, если на изображении имеется прямоугольный блок пикселов одного цвета, программа на языке PostScript для этого изображения будет содержать команды для помещения прямоугольника в определенное место изображения и заполнения его определенным цветом. Для хранения этих команд потребуется всего несколько битов. Когда принтер получает изображение, интерпретатор запускает программу и создает изображение. Таким образом, язык PostScript позволяет добиться сжатия данных за счет большего объема вычислений. Этот подход диа-

метрально противоположен рассматривавшемуся выше примеру оптимизации по скорости с поиском значения в таблице, но он также очень полезен, когда не хватает памяти или пропускной способности.

Другие примеры оптимизации включают структуры данных. Двойные связные списки занимают больше памяти, чем одинарные связные списки, зато они часто предоставляют более быстрый доступ к элементам списка. Хэш-таблицы занимают еще больше памяти, но еще более ускоряют поиск. Короче говоря, при оптимизации участка программы следует уделить особое внимание проблеме компромисса между занимаемой памятью и скоростью выполнения программы.

Кэширование

Хорошо известным методом повышения производительности является кэширование. Оно может применяться каждый раз, когда с большой вероятностью можно предсказать, что много раз потребуется один и тот же результат. Общий метод заключается в том, чтобы выполнить всю работу в первый раз, а затем сохранить результат в кэше. При последующих попытках в первую очередь будет проверяться кэш. Если результат находится в кэше, то нужно всего лишь достать его оттуда. В противном случае необходимо проделать всю работу сначала.

Мы уже наблюдали использование кэша в файловой системе, где он хранит некоторое количество недавно использовавшихся блоков диска, что позволяет избежать обращения к диску при чтении блока. Однако кэширование может также применяться и для других целей. Например, обработка путей к файлам отнимает удивительно много процессорного времени. Рассмотрим снова пример из системы UNIX, показанный на рис. 6.35. Чтобы найти файл */usr/ast/mbox*, потребуется выполнить следующие обращения к диску:

1. Прочитать i-узел корневого каталога (i-узел 1).
2. Прочитать корневой каталог (блок 1).
3. Прочитать i-узел каталога */usr* (i-узел 6).
4. Прочитать каталог */usr* (блок 132).
5. Прочитать i-узел каталога */usr/ast* (i-узел 26).
6. Прочитать каталог */usr/ast* (блок 406).

Чтобы просто определить номер i-узла искомого файла, нужно как минимум шесть раз обратиться к диску. Если размер файла меньше размера блока (например, 1024 байт), то, чтобы прочитать содержимое файла, нужно восемь обращений к диску.

В некоторых операционных системах обработка путей файлов оптимизируется при помощи кэширования пар (путь, i-узел). Например, на рис. 6.35 кэш будет содержать первые три записи табл. 12.1 после обработки пути */usr/ast/mbox*. Последние три записи попадают в таблицу после обработки других путей.

Когда файловая система должна найти файл по пути, обработчик путей сначала обращается к кэшу и ищет в нем самую длинную подстроку, соответствующую обрабатываемому пути. Если обрабатывается путь */usr/ast/grants/stw*, кэш отвечает, что номер i-узла каталога */usr/ast* равен 26, так что поиск может быть начат с этого места и количество обращений к диску может быть уменьшено на четыре.

Таблица 12.1. Часть кэша i-узлов для рис. 6.35

Путь	Номер i-узла
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/papers.ps	85

Недостаток кэширования путей состоит в том, что соответствие имени файла номеру его i-узла не является постоянным. Представьте, что файл */usr/ast/mbox* удаляется и его i-узел используется для другого файла, владельцем которого может быть другой пользователь. Затем файл */usr/ast/mbox* создается снова, но на этот раз он получает i-узел с номером 106. Если не предпринять специальных мер, запись кэша будет указывать на неверный номер i-узла. Поэтому при удалении файла или каталога следует удалять из кэша запись, соответствующую этому файлу, а если удаляется каталог, то следует удалить также все записи для содержавшихся в этом каталоге файлов и подкаталогов[1].

Кэшироваться могут не только блоки дисков и пути к файлам. Можно кэшировать также i-узлы. Если для обработки прерываний используются временные потоки, для каждого из них требуется стек и некоторый дополнительный механизм. Эти использовавшиеся ранее потоки также можно кэшировать, так как обновить уже использовавшийся поток легче, чем создать новый (применение кэша позволяет избежать необходимости в выделении новому процессу памяти). Кэширование может применяться почти для всего, что трудновоспроизводимо.

Подсказки

Элементы кэша всегда должны быть корректны. Поиск в кэше может завершиться неудачей, но если элемент найден, то его корректность гарантируется, поэтому найденный элемент может использоваться без дополнительных хлопот. В некоторых системах бывает удобным содержать таблицу **подсказок**. Подсказки представляют собой предложения решений, но их корректность не гарантируется. Обращающийся к этой таблице процесс должен сам проверять корректность результата.

Хорошо известным примером подсказок являются указатели URL, содержащиеся в web-страницах. Когда пользователь щелкает мышью на ссылке, он не получает гарантии, что соответствующая web-страница находится там, куда указывает URL. В самом деле, может оказаться, что требующаяся страница удалена несколько лет назад. Таким образом, информация, содержащаяся на web-странице, представляет собой всего лишь подсказку.

Подсказки также используются при работе с удаленными файлами. Информация, содержащаяся в подсказке, сообщает нечто об удаленном файле, например его местонахождение. Однако, возможно, этот файл уже удален или перемещен в другое место, поэтому всегда требуется проверка корректности подсказки.

[1] Во всех файловых системах удаляться могут только пустые каталоги. — *Примеч. перев.*

Использование локальности

Процессы и программы действуют не случайным образом. Они оказываются в значительной степени локальными как во времени, так и в пространстве, и эта информация может быть использована различными способами для улучшения производительности. Один хорошо известный пример пространственной локальности заключается в том факте, что процессы не прыгают произвольным образом в пределах своего адресного пространства. Как правило, за фиксированный интервал времени они используют относительно небольшое количество страниц. Страницы, активно используемые процессом, могут рассматриваться как рабочий набор процесса. А операционная система может гарантировать, что этот рабочий набор находится в памяти, когда процесс получает управление, тем самым снижается количество страничных прерываний.

Принцип локальности также применим для файлов. Когда процесс выбирает конкретный рабочий каталог, многие из его последующих файловых обращений, скорее всего, будут относиться к файлам, расположенным в этом каталоге. Производительность можно повысить, если поместить все файлы каталога и их i-узлы близко друг к другу на диске. Именно этот принцип лежит в основе файловой системы Berkeley Fast File System [231].

Другой областью, в которой локальность играет важную роль, является планирование потоков в мультипроцессорах. Как было показано в главе 8, один из методов планирования потоков заключается в том, чтобы попытаться запустить каждый поток на том центральном процессоре, на котором он работал в прошлый раз, в надежде, что какие-нибудь из его блоков все еще находятся в кэше.

Оптимизируйте общий случай

Часто бывает полезно различать наиболее частый случай и наименее вероятный случай и обращаться с ними по-разному. Обычно различные случаи обрабатываются совершенно различными программами. Важно, чтобы частый случай работал быстро. От алгоритма для редко встречающегося случая достаточно добиться корректной работы.

В качестве первого примера рассмотрим вхождение в критическую область. В большинстве случаев процессу будет удаваться вход в критическую область, особенно если внутри этой области процессы не проводят много времени. Операционная система Windows 2000 использует это преимущество, предоставляя вызов Win32 API `EnterCriticalSection`, который является атомарной функцией, проверяющей флаг в режиме пользователя (с помощью команды процессора `TSL` или ее эквивалента). Если тест проходит успешно, процесс просто входит в критическую область, для чего не требуется обращения к ядру. Если же результат проверки отрицательный, библиотечная процедура выполняет на семафоре операцию `down`, чтобы заблокировать процесс. Таким образом, в нормальном случае обращение к ядру не требуется.

В качестве второго примера рассмотрим установку будильника (использующего сигналы UNIX). Если в текущий момент ни один будильник не заведен, то про-

сто создается запись и помещается в очередь таймеров. Однако если будильник уже заведен, его следует найти и удалить из очереди таймера. Так как системный вызов alarm не указывает, установлен ли уже будильник, система должна предполагать худшее, то есть что он уже заведен. Однако в большинстве случаев будильник не будет заведен, и поскольку удаление существующего будильника представляет собой дорогое удовольствие, то следует различать эти два случая.

Один из способов достижения этой цели заключается в том, чтобы хранить бит в таблице процессов, указывающий, заведен ли будильник. Если бит сброшен, то программа следует по простому пути (просто добавляется новая очередь таймера без какой-либо проверки). Если бит установлен, то очередь таймера требует проверки.

Управление проектом

Многие программисты являются вечными оптимистами. Они полагают: чтобы написать программу, нужно всего лишь поскорее сесть за клавиатуру и начать набивать символы. Вскоре после этого появится полностью законченная отлаженная программа. Очень большие программы таким способом написать невозможно. В следующих разделах мы кратко обсудим вопросы управления большими программными проектами, особенно управления большими системными проектами.

Мифический человеко-месяц

В своей классической книге Фред Брукс, один из разработчиков системы OS/360, занявшийся впоследствии научной деятельностью, рассматривает вопрос, почему так трудно построить большую операционную систему [44, 46]. Когда большинство программистов встречаются с его утверждением, что специалисты, работающие над большими проектами, могут за год произвести всего лишь 1000 строк отлаженного кода, они удивляются, не прилетел ли профессор Брукс из космоса, с планеты Баг. В конце концов, большинство из них помнит, как они создавали программу из 1000 строк всего за одну ночь. Как же этот объем исходного текста может составлять годовую норму для любого программиста, чей IQ превышает 50?

Брукс отмечает, что большие проекты, в которых задействованы сотни программистов, принципиально отличаются от небольших проектов и что результаты, достигнутые при работе над небольшим проектом, нельзя переносить на большой проект. В большом проекте огромное количество времени тратится на планирование того, как разделить работу на отдельные модули. При этом нужно детально расписать работу модулей и интерфейсы к ним, а также попытаться представить себе, как эти модули взаимодействуют, причем до того, как начнется само программирование. Затем модули по отдельности создаются и отлаживаются. Наконец, модули собираются вместе и вся система в целом тестируется. Как правило, при этом собранная из работающих по отдельности модулей система работать не хочет,

и после сборки и запуска немедленно рушится. Брукс оценивает количество работ следующим образом:

1/3 планирование;

1/6 кодирование;

1/4 тестирование модулей;

1/4 тестирование системы.

Другими словами, собственно написание программы представляет собой самую простую часть проекта. Самым сложным оказывается решить, какими должны быть модули, а также заставить эти модули корректно общаться друг с другом. В небольшой программе, создаваемой одним программистом, планирование составляет как раз наиболее легкую часть.

Заголовком книги Брукс обращает внимание читателя на собственное утверждение о том, что люди и время не взаимозаменяемы. Такой единицы, как человеко-месяц, в программировании не существует. Если в проекте участвуют 15 человек, и на всю работу у них уходит 2 года, то отсюда не следует, что 360 человек справятся с этой работой за один месяц, и вряд ли 60 человек выполнят эту работу за 6 месяцев.

У этого явления есть три причины. Во-первых, работа не может быть полностью разделена. До тех пор пока не будет закончено планирование и не будет определено, какие модули нужны, а также какими будут интерфейсы, никакое программирование не может даже начаться. При двухлетнем проекте одно лишь планирование может занять 8 месяцев.

Во-вторых, чтобы полностью использовать большое число программистов, работу следует разделить на большое количество модулей, чтобы всех обеспечить работой. Поскольку потенциально каждый модуль взаимодействует с каждым модулем, число рассматриваемых пар модуль-модуль растет пропорционально квадрату от числа модулей, то есть квадрату числа программистов. Поэтому большие проекты с увеличением числа программистов очень быстро выходят из-под контроля. Тщательные измерения 63 программных проектов подтвердили, что зависимость времени выполнения проекта от количества программистов далеко не так проста, как можно предположить, исходя из концепции человеко-месяцев [37].

В-третьих, процесс отладки в большой степени является последовательным. Если усадить за решение проблемы вместо одного отладчика десятерых, это не поможет обнаружить ошибку в программе в десять раз быстрее. На самом деле десять отладчиков, вероятно, даже будут работать медленнее одного, так как они будут тратить очень много времени на разговоры друг с другом.

Брукс подводит итоги своего опыта знакомства с большими проектами, формулируя следующий закон (закон Брукса):

Добавление к программному проекту на поздней стадии дополнительных людских сил приводит к увеличению сроков выполнения проекта.

Недостаток введения в проект новых людей состоит в том, что их необходимо обучать, модули нужно делить заново, чтобы их число соответствовало числу программистов, требуется провести множество собраний, чтобы скоординировать работу отдельных групп и программистов и т. д. Абдель-Хамид и Мэдник [1]

получили экспериментальное подтверждение этого закона. Слегка фривольный вариант закона Брукса звучит так:

Если собрать девять рожениц в одной комнате, то они не родят через один месяц.

Структура команды

Коммерческие операционные системы представляют собой большие программные проекты, которые обязательно требуют участия в работе больших команд разработчиков. Уровень программистов имеет чрезвычайно высокое значение. Уже десятки лет известно, что сильнейшие программисты в десять раз продуктивнее плохих программистов [284]. Неприятность заключается в том, что когда вам нужны 200 программистов, сложно найти 200 программистов высочайшей квалификации. Вам придется согласиться на команду, состоящую из программистов самого различного уровня.

Что также важно в крупном проекте, программном или любом другом, — это необходимость архитектурного соответствия. Нужно, чтобы один человек контролировал всю конструкцию. В качестве примера Брукс приводит кафедральный собор в Реймсе, постройка которого заняла десятки лет, но архитекторы, сменявшие друг друга, не поддались искушению внести в проект собственные идеи и сумели сохранить изначальный архитектурный план. В результате было получено архитектурное соответствие, которого не удалось достичь в других соборах Европы.

В 70-е годы Харлан Миллс попытался объединить наблюдения о различиях в уровнях квалификации программистов с необходимостью архитектурного соответствия и ввел понятие **команды главного программиста** [17]. Его идея заключалась в том, чтобы организовать программистов в нечто подобное хирургической группы, в противоположность бригаде забойщиков свиней. В отличие от последней команды, в которой у каждого работника по ножу и каждый рубит им направо и налево как сумасшедший, в хирургической группе только один хирург держит в руке скальпель. Все остальные члены группы лишь ассистируют ему. Предложенная Миллсом структура команды из 10 разработчиков показана в табл. 12.2.

Таблица 12.2. Предложенная Миллсом структура команды из 10 разработчиков

Должность	Обязанности
Главный программист	Занимается архитектурным дизайном и пишет программу
Второй пилот	Помогает главному программисту и служит резонатором
Администратор	Управляет людьми, бюджетом, пространством, оборудованием, отчетами и т. д.
Редактор	Редактирует документацию, которую должен писать главный программист
Секретари	Администратору и редактору нужны секретари
Архивариус	Следит за состоянием архивов программ и документации
Инструментальщик	Снабжает главного программиста необходимыми инструментами
Тестер	Тестирует программы главного программиста
Эксперт по языкам	(Работает неполный рабочий день). Может дать совет главному программисту по языкам программирования

Такая структура команды была предложена тридцать лет назад. С тех пор кое-что изменилось (так, например, исчезла надобность в консультанте по языкам — язык С проще, чем PL/1). Но необходимость в централизованном управлении осталась. По-прежнему управлять всей схемой должен всего один человек. Хотя использование компьютеров позволяет уменьшить штат помощников, но сама идея все еще остается в силе.

Любой большой проект должен быть организован иерархически. На нижнем уровне располагается множество маленьких групп, каждую из которых возглавляет главный программист. На следующем уровне группы команд должны координироваться менеджером. Как показывает опыт, каждый управляемый менеджером человек обходится ему в 10 % рабочего времени, поэтому для каждой группы из 10 команд требуется отдельный менеджер. Этими менеджерами также нужно управлять и т. д., до вершины дерева.

Брукс отмечает, что плохие новости плохо распространяются вверх по дереву. Джерри Зальтцер из Массачусетсского технологического института назвал этот эффект **диодом плохих новостей**. Ни один программист или менеджер не хочет сообщать своему боссу, что проект на 4 месяца отстает от графика и не имеет шансов быть выполненным в срок из-за тысячелетней традиции отрубания головы посланнику, принесшему дурную новость. В результате старший менеджер, как правило, не имеет никакого представления о состоянии проекта. Когда становится очевидно, что в срок проект выполнен быть не может, старший менеджер нанимает дополнительных людей, после чего в действие вступает закон Брукса.

На практике большие компании, у которых накоплен значительный опыт в области производства программного обеспечения и которые знают, что произойдет, если оно создается бессистемно, по крайней мере пытаются действовать правильно. Небольшие, новые компании, изо всех сил спешащие прорваться на рынок, напротив, не всегда заботятся о том, чтобы аккуратно производить свое программное обеспечение. Эта спешка часто приводит к далеко не оптимальным результатам.

Ни Брукс, ни Миллс не смогли предвидеть роста производства открытых программных средств. Хотя в этой области имеется ряд успехов, открытые программные средства до сих пор рассматриваются, как если бы это была жизнеспособная модель производства большого количества качественного программного обеспечения, стоит только пройти эффекту новизны. Вспомните, что в первые годы после изобретения радио в эфире доминировали радиолюбители, вскоре им пришлось уступить коммерческому радио, а затем коммерческому телевидению. Следует заметить, что наиболее удачные проекты открытых программных средств, несомненно, использовали модель главного программиста, то есть всем проектом управлял один человек (например, Линус Торвальдс, руководивший разработкой ядра операционной системы Linux, и Ричард Столман, направлявший процесс создания компилятора GNU C).

Роль опыта

Наличие опытных разработчиков является критичным при проектировании операционной системы. Брукс указывает, что большинство ошибок допускается ни при программировании, а на стадии проекта. Программисты правильно делают то,

что им велят делать. Но если то, что им велели, было неверно, то никакое тестовое программное обеспечение не сможет поймать ошибку неверно составленной спецификации.

Решение, предложенное Бруксом, заключается в отказе от классической модели разработки (рис. 12.4, *а*) и в использовании модели, показанной на рис. 12.4, *б*. Принцип состоит в том, чтобы сначала написать главный модуль программы, который просто вызывает процедуры верхнего уровня. Вначале эти процедуры представляют собой заглушки. Начиная уже с первого дня система будет транслироваться и запускаться, хотя делать она ничего не будет. Постепенно в систему будут устанавливаться модули. Результат применения такого метода заключается в том, что сборка системы проверяется постоянно, поэтому ошибки в проекте обнаруживаются значительно раньше. Таким образом, процесс обучения на собственных ошибках также начинается значительно раньше.

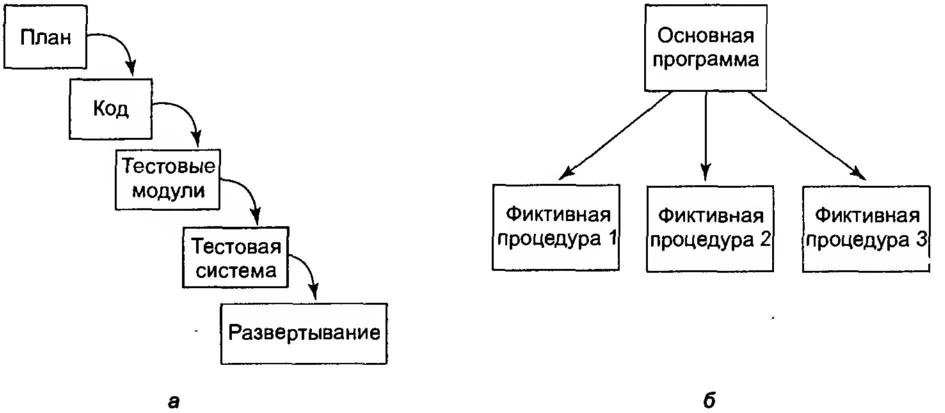


Рис. 12.4. Традиционное поэтапное проектирование программного обеспечения (*а*); альтернативный метод создания работающей уже с первого дня системы (*б*)

Неполные знания представляют собой опасную вещь. В своей книге Брукс описывает явление, названное им **эффектом второй системы**. Часто первый продукт, созданный группой разработчиков, является минимальным, так как они опасаются, что он не будет работать вообще. Поэтому они опасаются помещать в первый выпуск программного обеспечения много функций. Если проект оказывается удачным, они создают следующую версию программного обеспечения. Воодушевленные собственным успехом, во второй раз разработчики включают в систему все погремушки и побрякушки, намеренно не включенные в первый выпуск. В результате система раздувается и ее производительность снижается. От этой неудачи команда разработчиков трезвеет и при выпуске третьей версии снова соблюдает осторожность.

Это наблюдение отчетливо видно на примере пары систем CTSS-MULTICS. Операционная система CTSS была первой универсальной системой разделения времени и ее успех был огромен, несмотря на минимальную функциональность системы. Создатели операционной системы MULTICS, преемницы CTSS, были слишком амбициозны, за что и поплатились. Сами идеи были неплохи, но новых функций добавилось слишком много, что сказалось на низкой производительнос-

ти системы, страдавшей этим недугом в течение долгих лет и так и не получившей коммерческого успеха. Третьей в этой линейке была операционная система UNIX, разработчики которой проявили значительно большую осторожность и в результате добились существенно больших успехов.

Панацеи нет

Помимо книги «Мифический человеко-месяц», Брукс написал также статью «No Silver Bullet» (Панацеи нет), получившую широкий резонанс [45]. В ней он доказывал, что в ближайшие десять лет ни одно средство, поисками которого в те времена занималось множество людей, не приведет к существенному (на порядок) увеличению производительности в создании программного обеспечения. Как показывает опыт последних лет, он был прав.

Среди предлагаемых в то время чудодейственных средств были улучшенные языки высокого уровня, объектно-ориентированное программирование, искусственный интеллект, экспертные системы, автоматическое программирование, графическое программирование, верификация программ и программное окружение. Возможно, в следующие десять лет мы увидим нечто радикальное, но не исключено, что нам придется довольствоваться постепенными последовательными улучшениями.

Тенденции в проектировании операционных систем

Предсказывать всегда трудно, особенно будущее. Например, в 1899 году Чарльз Дьюэл, возглавлявший тогда Бюро патентов США, предложил тогдашнему президенту США Мак-Кинли ликвидировать патентное бюро (а также и рабочее место Чарльза Дьюэла!), поскольку, как он писал, «все, что можно было изобрести, уже изобретено» [58]. Тем не менее прошло всего несколько лет, и на пороге патентного бюро показался Томас Эдисон с заявками на электрические лампы, фонограф и кинопроектор. Сменим батарейки в нашем кристальном шаре и попытаемся угадать, что станет с операционными системами в ближайшем будущем.

Операционные системы с большим адресным пространством

По мере того как на смену 32-разрядным машинам приходят 64-разрядные, становится возможным главное изменение в строении операционных систем. 32-разрядное адресное пространство на самом деле не так уж велико. Если попытаться разделить 2^{32} байт на всех жителей Земли, то каждому достанется менее одного байта. В то же время 2^{64} примерно равно 2×10^{19}. При этом каждому жителю планеты в 64-разрядном адресном пространстве можно выделить фрагмент размером в 3 Гбайт.

Что можно сделать с адресным пространством в 2×10^{19} байт? Для начала мы можем отказаться от концепции файловой системы. Вместо этого все файлы можно

постоянно хранить в памяти (виртуальной). В конце концов, в ней достаточно места для более чем миллиарда полнометражных фильмов, сжатых до 4 Гбайт.

Другая возможность заключается в использовании перманентных объектов. Объекты могут создаваться в адресном пространстве и храниться в нем до тех пор, пока не будут удалены все ссылки на объект, после чего сам объект автоматически удаляется. Такие объекты будут сохраняться в адресном пространстве даже после выключения и перезагрузки компьютера. Чтобы заполнить все 64-разрядное адресное пространство, нужно создавать объекты со скоростью 100 Мбайт/с в течение 5000 лет. Разумеется, для хранения такого количества данных потребуется очень много дисков, но впервые в истории ограничивающим фактором стали физические возможности дисков, а не адресное пространство.

При большом количестве объектов в адресном пространстве становится интересным позволить нескольким процессам работать одновременно в одном адресном пространстве, чтобы упростить совместное использование объектов. Применение такой схемы, разумеется, приведет к появлению операционных систем, сильно отличающихся от существующих в настоящий момент. Некоторые соображения об этой концепции содержатся в [60].

Еще один системный аспект, который придется пересмотреть при введении 64-разрядных адресов, это виртуальная память. При 2^{64} байт виртуального адресного пространства и 8-килобайтных страницах у нас будет 2^{51} страниц. Работать с обычными таблицами страниц такого размера будет непросто, поэтому потребуется другое решение. Возможно использование инвертированных таблиц страниц, однако также предлагались и другие идеи [321]. В любом случае появление 64-разрядных операционных систем создает новую большую область исследований.

Сеть

Современные операционные системы разрабатывались для автономных компьютеров. Сети были разработаны позднее, и доступ к ним главным образом предоставляется при помощи специальных программ и протоколов, таких как web-браузеры, FTP или telnet. В будущем, возможно, сети будут составлять основу всех операционных систем. Автономный компьютер, не подключенный к сети, будет столь же редким явлением, как и телефон, не подключенный к линии. И, скорее всего, соединения с пропускной способностью в десятки и сотни мегабит в секунду станут нормой.

Чтобы приспособиться к этому сдвигу парадигм, операционным системам придется измениться. Различие между локальными данными и удаленными данными может размыться, так как практически никого не будет беспокоить, где фактически хранятся данные. С любыми данными компьютер сможет работать, как с локальными. В системе NFS это уже в определенном смысле так, но, похоже, эта тенденция будет продолжена и расширена, и в этой области будет достигнута более высокая степень интеграции.

Доступ к Всемирной паутине, для которого в настоящий момент требуются специальные программы (браузеры), также может стать полностью интегрированным в операционную систему. Web-страницы, возможно, станут стандартным

способом хранения информации, а эти страницы могут содержать очень широкий спектр данных нетекстового формата, включая аудио, видео, программы и т. д., и всеми этими данными операционная система будет управлять, как своими основными данными.

Параллельные и распределенные системы

Другой новой областью являются параллельные и распределенные системы. Современные операционные системы для мультипроцессоров и многокомпьютерных систем представляют собой просто стандартные однопроцессорные операционные системы с небольшими изменениями в устройстве планировщика, обеспечивающими несколько лучшую поддержку параллелизма. В будущем, возможно, у нас будут операционные системы, в которых параллелизму будет предоставлено центральное место. Серьезным дополнительным стимулом к этому станет возможное использование мультипроцессорных схем для настольных компьютеров. В результате может появиться множество прикладных программ, специально разработанных для работы на мультипроцессорах, а также необходимость в лучшей поддержке этой работы со стороны операционной системы.

Многокомпьютерные системы, скорее всего, в ближайшие годы будут доминировать среди научных и инженерных суперкомпьютеров, но операционные системы для них все еще остаются крайне примитивными. Для помещения процессов, балансировки загрузки и обмена информацией требуется много работы.

Современные распределенные системы часто строятся как промежуточное программное обеспечение, так как существующие операционные системы не предоставляют распределенным приложениям всех необходимых функций. Возможно, при проектировании будущих операционных систем будут учитываться распределенные системы, поэтому все необходимые функции будут присутствовать в операционной системе с самого начала.

Мультимедиа

Мультимедийные системы стали восходящей звездой в компьютерном мире. Никого не удивит, если компьютеры, стереоустановки, телевизоры и телефоны будут объединены в одно устройство, обеспечивающее воспроизведение высококачественного звука и видеоизображения, а также подключенного к высокоскоростной сети, что обеспечит быструю загрузку требуемых файлов. Операционные системы для этих устройств или даже для автономных аудио- и видеоустройств должны существенно отличаться от современных операционных систем. В частности, потребуются гарантии реального времени, и они составят основу устройства системы. Кроме того, пользователи окажутся очень недовольными, если операционную систему их телевизоров придется перезагружать через каждый час, поэтому к программному обеспечению будут предъявляться более высокие требования по качеству и устойчивости к сбоям. К тому же размер мультимедийных файлов, как правило, очень велик, поэтому от файловой системы требуется способность эффективной работы с ними.

Компьютеры на батарейках

Без всякого сомнения, мощные настольные персональные компьютеры, вероятно, с 64-разрядным адресным пространством, с соединением с высокоскоростной сетью, несколькими процессорами и высококачественными изображением и звуком, станут нормальным явлением. Их операционные системы должны существенно отличаться от современных систем, чтобы обеспечить поддержку всех этих требований. Однако еще более быстрый сегмент рынка составляют компьютеры, питающиеся от батарей, к которым относятся лэптопы, палмтопы, web-пады и различные телефонные гибриды. Некоторые из них поддерживают беспроводное соединение с внешним миром. Другие работают в автономном режиме, а к сети могут подключаться стационарно. Для них нужны операционные системы, отличающиеся от современных меньшими размерами, большей гибкостью и большей надежностью. Основу этих систем могут составить различные виды микроядерных и расширяемых систем.

Эти системы должны будут лучше современных систем поддерживать работу с полным соединением (то есть с использованием проводов), со слабым соединением (с использованием беспроводной связи) и автономную работу, включая накопление данных перед отключением от сети и проверку непротиворечивости данных перед тем, как снова подключиться к сети. Они также должны лучше справляться с проблемами мобильности (например, находить лазерный принтер, регистрироваться на нем и посылать ему файл по радио). Особое значение для этих систем имеет управление энергопотреблением, включая продолжительные диалоги между операционной системой и приложениями о том, сколько осталось энергии в батареях и как ее лучше всего использовать. Также может стать важной динамическая адаптация приложений к крошечным экранам. Наконец, для новых режимов ввода и вывода, включая ввод рукописного текста и речевой ввод, в операционной системе могут потребоваться новые методы, позволяющие повысить качество. Маловероятно, что операционная система для питаемого батареями, портативного беспроводного, управляемого голосом компьютера будет иметь много общего с системой, работающей на 64-разрядном мультипроцессоре с четырьмя центральными процессорами, с гигабитным оптоволоконным сетевым соединением. И, разумеется, будет бесчисленное множество гибридных машин, со своими собственными требованиями.

Встроенные системы

Последняя область, в которой будут плодиться и размножаться новые операционные системы, — это встроенные системы. Операционные системы в стиральных машинах, микроволновых печах, куклах, транзисторных (Интернет?) приемниках, MP3-проигрывателях, видеокамерах, лифтах и кардиостимуляторах будут отличаться ото всех перечисленных выше и, скорее всего, друг от друга. Каждая из них должна быть тщательно скроена под ее конкретное приложение, так как маловероятно, что кому-либо придет в голову засунуть карту PCI в кардиостимулятор, чтобы превратить его в контроллер лифта. Поскольку во всех встроенных системах работает только ограниченный набор программ, известных заранее, можно запретить оптимизацию в универсальных системах.

Расширяемые операционные системы (например, Paramecium и Exokernel) оказываются многообещающей идеей для встроенных систем. Их можно сделать легковесными или тяжеловесными, в зависимости от потребностей конкретного приложения, при этом, правда, следует добиваться непротиворечивости между приложениями. Поскольку встроенные системы будут производиться сотнями миллионов, это станет главным рынком для операционных систем.

Резюме

Проектирование операционных систем начинается с определения их задач. Интерфейс должен быть простым, полным и эффективным. Он должен обладать ясной парадигмой пользовательского интерфейса, парадигмой исполнения и парадигмой данных.

Система должна быть хорошо структурированной, для чего может быть использована одна из нескольких известных технологий, таких как многоуровневые системы или архитектуры клиент-сервер. Внутренние компоненты должны быть ортогональными друг к другу. Кроме того, следует четко отделить политику от механизма. Следует также уделить существенное внимание таким вопросам, как выбор между статическими или динамическими структурами данных, именование, время связывания, а также порядок реализации модулей.

Производительность является важным вопросом, но следует тщательно выбирать способ оптимизации, чтобы не нарушить структуру операционной системы. Часто имеет смысл заниматься оптимизацией по скорости или по занимаемой памяти, кэшированием, подсказками, использовать локальность, а также оптимизировать общий случай.

Создание системы группой из двух-трех человек отличается от разработки большой системы командой из 300 сотрудников. В последнем случае в успехе или неуспехе проекта главную роль играют такие вопросы, как структура команды и управление проектом.

Наконец, в ближайшие годы операционные системы должны будут измениться, чтобы соответствовать новым тенденциям и новым требованиям. К ним могут относиться 64-разрядное адресное пространство, высокоскоростное сетевое соединение, настольные мультипроцессоры, мультимедиа, переносные компьютеры с беспроводной связью, а также огромное количество встроенных систем. Ближайшие годы будут весьма интересными для проектировщиков операционных систем.

Вопросы

1. Закон Мура описывает явление экспоненциального роста, сходного с ростом популяций видов животных, помещенных в новую среду с изобилием пищи и отсутствием естественных врагов. В природе кривая экспоненциального роста в конце концов становится сигмоидальной, асимптотически стремящейся к некоему пределу, когда обнаруживается ограниченность запасов пищи или хищники приспосабливаются к новой добыче. Обсудите факторы, способные ограничить рост производительности компьютерной аппаратуры.

2. В листинге 12.1 показаны две парадигмы, алгоритмическая и движимая событиями. Какую парадигму проще всего применить для каждой из следующих типов программ:
   а) компилятор;
   б) программа редактирования фотографий;
   в) программа составления платежной ведомости.
3. На некоторых ранних моделях компьютера Apple Macintosh программа графического интерфейса пользователя располагалась в ПЗУ. Почему?
4. Иерархические имена файлов всегда начинаются с вершины дерева. Например, имя файла записывается как */usr/ast/books/mos2/chap-12*, а не *chap-12/mos2/books/ast/usr*. Имена DNS, напротив, начинаются на дне дерева и двигаются вверх. Есть ли какая-либо фундаментальная причина для этого отличия?
5. Правило Корбато утверждает, что система должна предоставлять минимальный механизм. Ниже приводится список вызовов стандарта POSIX, также присутствовавших в системе UNIX Version 7. Какие из них являются избыточными, то есть могут быть удалены без потери функциональности, так как та же работа может быть выполнена при помощи простой комбинации остальных вызовов примерно с той же производительностью? `Access`, `alarm`, `chdir`, `chmod`, `chown`, `chroot`, `close`, `creat`, `dup`, `exec`, `exit`, `fcntl`, `fork`, `fstat`, `ioctl`, `kill`, `link`, `lseek`, `mkdir`, `mknod`, `open`, `pause`, `pipe`, `read`, `stat`, `time`, `times`, `umask`, `unlink`, `utime`, `wait`, и `write`.
6. Предположим, что уровни 3 и 4 на рис. 12.1 переставлены местами. Какие последствия окажет это на конструкцию системы?
7. В системе клиент-сервер, основанной на микроядре, микроядро занимается только передачей сообщений. Могут ли процессы пользователя, несмотря на это, создавать и использовать семафоры? Если да, то как? Если нет, почему нет?
8. Аккуратная оптимизация может повысить производительность системных вызовов. Рассмотрим случай, в котором каждые 10 мс выполняется один системный вызов. Среднее время вызова составляет 2 мс. Насколько ускорится процесс, занимавший 10 с, если ускорить выполнение системных вызовов в два раза?
9. Обсудите тему политики и механизма в контексте розничной торговли.
10. Операционные системы часто поддерживают два уровня именования: внешнее и внутреннее. В чем различие этих имен в плане
    а) длины;
    б) уникальности;
    в) иерархии.
11. Один из способов работы с таблицами, размер которых не известен заранее, заключается в том, чтобы сделать эти таблицы фиксированными, но когда одна таблица заполняется, заменять ее таблицей большего размера, копировать старые записи в новую таблицу, после чего память, занимаемая старой таблицей, освобождается. В чем преимущества и недостатки удвоения размеров таблицы по сравнению с увеличением размеров всего в 1,5 раза?

12. В листинге 12.2 флаг *found* используется для определения, был ли найден PID. Можно ли не сохранять флаг *found* в цикле, а просто проверять значение переменной *p* в конце цикла, чтобы определить, был ли найден процесс с данным идентификатором?
13. В листинге 12.3 различия между процессорами Pentium и UltraSPARC скрыты при помощи условной компиляции. Может ли использоваться тот же метод для того, чтобы скрыть различия между компьютером с процессором Pentium и жестким диском с интерфейсом IDE и компьютером с процессором Pentium и жестким диском с интерфейсом SCSI? Будет ли это удачной идеей?
14. Косвенность представляет собой метод увеличения гибкости алгоритма. Есть ли у этого метода недостатки, и если да, то какие?
15. Могут ли у реентерабельных процедур быть статические глобальные переменные? Аргументируйте свой ответ.
16. Макрос в листинге 12.4, *б*, очевидно, является более эффективным, чем процедура в листинге 12.4, *а*. Однако этот макрос труден для восприятия. Есть ли у него другие недостатки? Если да, то какие?
17. Допустим, нам нужен способ определить, является ли количество битов в 32-разрядном слове четным или нечетным. Разработайте алгоритм, позволяющий определять это как можно быстрее. При необходимости вы можете использовать для таблиц до 256 Кбайт ОЗУ. Напишите макрос, выполняющий данный алгоритм. *Дополнительное задание*: напишите процедуру, вычисляющую то же значение, перебирая 32 бит в цикле. Измерьте, во сколько раз макрос быстрее процедуры.
18. На рис. 12.3 показано, как файлы формата GIF используют 8-разрядные значения индексов в палитре цветов. Ту же идею можно использовать с 16-разрядной палитрой цветов. При каких обстоятельствах, если таковые вообще существуют, использование 24-разрядной палитры цветов может быть удачной идеей?
19. Один из недостатков формата GIF состоит в том, что изображение должно содержать палитру цветов, увеличивающую размер файла. При каком минимальном размере файла использование 8-разрядной палитры цветов не увеличит размеры файла? А для 16-разрядной палитры цветов?
20. В тексте было показано, как кэширование путей к файлам может существенно ускорить поиск файлов по имени. Другой иногда применяемый метод заключается в использовании демона, открывающего все файлы корневого каталога и постоянно хранящего их в открытом состоянии, чтобы их i-узлы постоянно присутствовали в памяти. Ускоряет ли данный метод поиск файлов по имени?
21. Даже если удаленный файл не был удален с того момента, когда была записана подсказка, он мог быть изменен с момента последнего доступа к нему. Какую еще информацию было бы полезно записывать?
22. Рассмотрим систему, накапливающую ссылки к удаленным файлам в виде подсказок, например, в формате (имя, удаленный хост, удаленное имя). Может случиться так, что удаленный файл будет удален, а затем заменен

другим файлом. При этом подсказка может указывать на неверный файл. Как можно снизить вероятность появления этой проблемы?

23. В тексте указывалось, что локальность часто можно использовать для увеличения производительности. Но рассмотрим случай, в котором программа читает входные данные из одного источника и постоянно выводит данные в один или несколько файлов. Может ли попытка использования локальности в файловой системе привести в данном случае к снижению производительности? Есть ли способ борьбы с этим?
24. Фред Брукс заявляет, что программист может написать за год всего 1000 строк отлаженного кода, однако первая версия системы MINIX (13 000 строк кода) была создана одним человеком менее чем за три года. Как вы объясните это несоответствие?
25. Основываясь на формуле Брукса о 1000 строк кода на программиста в год, оцените количество денежных средств, потраченных на создание операционной системы Windows 2000. Предположим, что программист обходится компании в 100 000 долларов в год (включая такие накладные расходы, как компьютеры, офисное пространство, секретарская поддержка и руководство). Правдоподобный ли получился ответ? Если нет, то какое из предположений могло быть неверно?
26. Память компьютеров становится все дешевле и дешевле, и уже можно представить себе компьютер с большой памятью, питающейся от батарей, вместо жесткого диска. При текущих ценах сколько может стоить такой персональный компьютер? Предположим, что для самой дешевой машины достаточно гигабайтного RAM-диска. Будет ли такая машина конкурентоспособной на рынке?
27. Назовите несколько особенностей обычной операционной системы, которые не нужны во встроенной системе, используемой внутри прибора.
28. Напишите на языке C процедуру, складывающую два заданных параметра с двойной точностью. Напишите процедуру, используя условную компиляцию, таким образом, чтобы эта процедура работала как на 16-разрядных, так и на 32-разрядных компьютерах.
29. Напишите программу, помещающую случайно сформированные короткие строки в массив, а затем находящую заданную строку в массиве при помощи (а) простого линейного поиска (метод грубой силы) и (б) более сложного метода по вашему выбору. Перекомпилируйте ваши программы для размеров массивов, варьирующихся от небольших до настолько больших, какие сможет поддержать ваша система. Оцените производительность обоих методов. При каком размере массивов производительность обоих методов будет одинаковой?
30. Напишите программу, моделирующую находящуюся в памяти файловую систему.

Библиография

В предыдущих 12 главах был рассмотрен широкий спектр вопросов. Цель этой главы заключается в том, чтобы помочь заинтересованным читателям продолжить изучение операционных систем. Первый раздел представляет собой список книг, рекомендованных для дальнейшего чтения. Второй раздел является алфавитным списком всех книг, на которые есть ссылки в данной книге.

Помимо этих списков литературы, в поисках новых статей по вопросам операционных систем стоит заглянуть в материалы очередного симпозиума Ассоциации по вычислительной технике (ACM — Association for Computing Machinery) по вопросам принципов операционных систем, проводимого ежегодно, а также в отчеты ежегодной международной конференции по распределенным вычислительным системам (IEEE). Представляют интерес и материалы симпозиума по проектированию и реализации операционных систем USENIX. Кроме того, интересные статьи об операционных системах печатаются в журналах *ACM Transactions on Computer Systems* и *Operating Systems Review*.

Литература, рекомендуемая для дальнейшего чтения

В следующих разделах будут даны некоторые рекомендации для дальнейшего чтения. В отличие от статей, упомянутых в разделах, озаглавленных «Исследования в области...» в тексте книги и посвященных текущим исследованиям, эти ссылки, по большей мере, представляют собой вступительные или учебные издания. Они могут использоваться для освещения материала, представленного в данной книге, с иной стороны.

Введение и общие труды

1. Kavi et al., «Computer Systems Research: The Pressure is On».

В каком направлении продвигаются исследования систем? Что является важным в настоящий момент? Как обстоит дело с разработкой прочных, предсказуемых систем? А с микросхемами, состоящими из миллиарда транзисторов, и со всемирными системами с миллиардом пользователей? В этой статье можно найти эти и другие вопросы, а также ответы на них.

2. Milojicic, «Operating Systems: Now and in the Future».

Представьте, что вы можете задать шести ведущим мировым экспертам серии вопросов по теме операционных систем и путям их развития. Получите ли вы одинаковые ответы? *Подсказка*: нет. В книге вы можете познакомиться с их мнениями.

3. Silberschatz et al., *Applied Operating System Concepts.*

Общее руководство по операционным системам. В нем обсуждаются процессы, управление хранением данных, распределенные системы и защита. Рассматриваются три конкретных примера: UNIX, Linux и Windows NT. Обложка целиком покрыта динозаврами. Какое отношение имеют они к операционным системам 2000-го года, неясно.

4. Stallings, *Operating Systems, 4th Ed.*

Еще один учебник по операционным системам. В нем описываются все традиционные темы, а также включено небольшое количество материала по распределенным системам.

5. Stevens, *Advanced Programming in the UNIX Environment.*

Эта книга сообщает, как написать программы на языке C, использующие интерфейс системных вызовов UNIX и стандартную библиотеку C. Примеры основаны на версиях системы UNIX System V Release 4 и 4.4BSD. Подробно описано отношение этих реализаций к стандарту POSIX.

Процессы и потоки

1. Andrews and Schneider, «Concepts and Notations for Concurrent Programming».

Учебный и обзорный материал по процессам и межпроцессному взаимодействию, в котором, среди прочего, описывается активное ожидание, семафоры, мониторы, передача сообщений и другие технологии. В этой статье также показывается, как эти понятия встроены в различные языки программирования.

2. Ben-Ari, *Principles of Concurrent Programming.*

Эта небольшая книга полностью посвящена проблемам межпроцессного взаимодействия. Среди прочих тем в отдельных главах обсуждаются взаимные исключения, семафоры, мониторы и задача обедающих философов.

3. Silberschatz et al., *Applied Operating System Concepts.*

Главы с 4 по 6 посвящены теме межпроцессного взаимодействия, включая планирование, критические области, семафоры, мониторы и классические проблемы межпроцессного взаимодействия.

Взаимоблокировка

1. Coffman et al., «System Deadlocks».

Эта книга представляет собой краткое введение во взаимоблокировки. В ней рассказывается о причинах их возникновения и способах их предотвращения или обнаружения.

2. Holt, «Some Deadlock Properties of Computer Systems».

Обсуждение взаимоблокировок. Холт вводит модель направленных графов, которую можно использовать для анализа некоторых тупиковых ситуаций.

3. Isloor and Marsland, «The Deadlock Problem: An Overview».

Учебное пособие по взаимоблокировкам, в котором особое внимание уделяется системам баз данных. Описывается множество моделей и алгоритмов.

Управление памятью

1. Denning, «Virtual Memory».

Классическая статья по многим вопросам виртуальной памяти. Деннинг был одним из пионеров в этой области. Он же является автором концепции рабочего набора.

2. Denning, «Working Sets Past and Present».

Хороший обзор многочисленных алгоритмов управления памятью и страничной подкачкой. Включена всеобъемлющая библиография.

3. Knuth, *The Art of Computer Programming*. Vol. 1.

В этой книге обсуждаются и сравниваются различные алгоритмы управления памятью, такие как «первый подходящий», «лучший подходящий» и т. д.

4. Silberschatz et al, *Applied Operating System Concepts*.

Главы 9 и 10 посвящены управлению памятью, включая свопинг, страничную подкачку и сегментацию. Упоминаются различные алгоритмы замещения страниц.

Ввод-вывод

1. Chen et al., «RAID: High Performance Reliable Secondary Storage».

Параллельное использование нескольких жестких дисков для ускорения ввода-вывода является современной тенденцией в профессиональных системах. Авторы обсуждают эту идею и изучают различные способы организации, сравнивая производительность, стоимость и надежность.

2. *Computer*, March 1994.

Этот выпуск журнала *Computer* содержит восемь статей по передовым технологиям ввода-вывода, в которых обсуждаются такие темы, как моделирование, накопители с высокой производительностью, кэширование, ввод-вывод для параллельных компьютеров и мультимедиа.

3. Geist and Daniel, «A Continuum of Disk Scheduling Algorithms».

В данной книге обсуждается обобщенный алгоритм планирования перемещений блока головок диска. Приводятся результаты всесторонних экспериментов и моделирования.

4. Gibson and Van Meter, «Network Attached Storage».

С появлением Интернета возросли потребности в больших накопителях для web-страниц, баз данных и других серверов. В результате было разработано множество схем присоединения к сети автономных запоминающих устройств. В данной

статье обсуждается несколько архитектурных решений, предназначенных для достижения этой цели.

5. Ng, «Advances in Disk Technology: Performance Issues».

Производительность системы может зависеть от различных факторов, таких как линейная плотность записи, скорость вращения диска, количество головок и количество секторов на дорожке. Эти вопросы, а также их влияние на производительность обсуждаются в этой легкой для чтения статье по технологии дисков.

6. Ruemmler and Wilkes, «An Introduction to Disk Drive Modeling».

Первая часть этой статьи посвящена современным дисковым накопителям и их внутренней работе, включая такие вопросы, как перемещение головок, зонирование, перекос дорожек, резервирование, кэширование, опережающее чтение и многое другое. Во второй части статьи описывается моделирование дисковых накопителей.

7. Walker and Cragon, «Interrupt Processing in Concurrent Processors».

Реализация точных прерываний на суперскалярных компьютерах представляет собой чрезвычайно перспективную область исследований. Хитрость заключается в том, чтобы преобразовать состояние процессора в последовательную форму и сделать это быстро. В данной статье обсуждаются различные вопросы устройства компьютеров и возможные компромиссные решения.

Файловые системы

1. Harbron, *File Systems.*

Книга по устройству файловых систем, приложениям и производительности. Описываются как структура, так и алгоритмы.

2. McKusick et al., «A Fast File System for UNIX».

Файловая система для UNIX была полностью переделана для версии 4.2 BSD. В этой статье описывается устройство новой файловой системы с особым акцентом на ее производительности.

3. Silberschatz et al., *Applied Operating System Concepts.*

Глава 11 посвящена теме файловых систем. Среди прочих тем в ней описываются операции с файлами, методы доступа, каталоги и вопросы реализации.

4. Stallings, *Operating Systems, 2nd Ed.*

В главе 14 содержится существенное количество материала о безопасном окружении, а также о хакерах, вирусах и других угрозах.

Мультимедийные операционные системы

1. *ACM Computing Surveys*, Dec. 1995.

Этот выпуск *ACM Computing Surveys* содержит 21 краткую статью по различным аспектам мультимедиа, варьирующихся от низкоуровневых технических вопросов до вопросов прикладного уровня.

2. *Computer*, May 1995.

Темой этого выпуска журнала *Computer* является мультимедиа. Этой теме журнал уделил целых шесть статей. После краткого вступления статьи рассказывают об интерактивном телевидении, мультимедийных серверах, об управлении исследованиями и о приложениях в медицине и обучении.

3. Lee, «Parallel Video Servers: A Tutorial».

Многие организации хотят предоставлять видео по заказу, в результате чего возникает потребность в масштабируемых, отказоустойчивых параллельных видеосерверах. Здесь обсуждаются главные вопросы, касающиеся их создания и включающие архитектуру серверов, чередующиеся наборы, политику размещения, балансировку нагрузки, избыточность, протоколы и синхронизацию.

4. Leslie et al., «The Design and Implementation of an Operating System to Support Distributed Multimedia Applications».

Многие попытки реализации мультимедиа основывались на добавлении новых функций к существующей операционной системе. Альтернативный подход, описанный в данной статье, заключается в том, чтобы начать все сначала и создать с нуля новую операционную систему для мультимедиа, без необходимости в обратной совместимости с чем бы то ни было. В результате получается операционная система, принципиально отличная по своему устройству от существующих.

5. Reddy, «I/O Issues in a Multimedia System».

Когда говорят о производительности компьютера, как правило, имеют в виду производительность центрального процессора. Однако для мультимедиа производительность ввода-вывода по меньшей мере так же важна. Именно этой теме и посвящена данная статья. Среди прочих вопросов в ней обсуждаются планирование дисков, вопрос зависимости производительности системы от количества оперативной памяти, а также управление доступом.

6. Sitaram and Dan, «Multimedia Servers».

У мультимедийных серверов есть множество отличий от обычных файловых серверов. Авторы подробно обсуждают эти различия, особенно в области планирования, запоминающей подсистемы и кэширования.

Многопроцессорные системы

1. Ahmad, «Gigantic Clusters: Where Are They and What Are They Doing?».

Чтобы понять идею, лежащую в основе современных больших многокомпьютерных систем, стоит прочитать эту статью. В ней описывается сама идея, а также рассматриваются некоторые наиболее крупные из работающих сегодня систем. Учитывая действие закона Мура, можно смело предполагать, что упоминаемые в этой статье размеры будут удваиваться примерно каждые 2 года.

2. Bhoedjang et al., «User-Level Network Interface Protocols».

Все увеличивающееся количество многомашинных систем для повышения производительности работают с сетевой интерфейсной картой в пространстве пользователя. В результате подобного подхода появляется множество вопросов

проектирования, одиннадцать из которых обсуждаются в данной статье. Также сравниваются несколько существующих систем.

3. *Computer*, Dec 1996.

В этом выпуске журнала *Computer* содержится восемь статей о мультипроцессорах. Одна из них является учебной по вопросу семантики совместно используемой памяти, но остальные восемь посвящены мультипроцессорным приложениям и производительности.

4. Dubois et al., «Synchronization, Coherence, and Event Ordering in Multiprocessors».

Учебная статья по вопросам синхронизации в мультипроцессорных системах с общей памятью. Однако некоторые идеи в равной мере применимы также к однопроцессорным системам и системам с распределенной памятью.

5. Kwok and Ahmad. «Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors».

Оптимальное планирование заданий на многомашинной системе или мультипроцессоре возможно, когда характеристики всех заданий известны эаранее. Проблема заключается в том, что расчет оптимального планирования занимает слишком много времени. В данной статье авторы обсуждают и сравнивают 27 известных алгоритмов для различных способов решения этой проблемы.

6. Langendoen et al., «Models for Asynchronous Message Handling».

Производительность многомашинных систем в значительной степени зависит от производительности системы передачи сообщений, особенно от того, как обрабатываются поступающие сообщения. Основными вариантами являются активные сообщения, функции обратного вызова и временные потоки. В данной статье авторы описывают все три способа обработки сообщений, а также сравнивают результаты экспериментов, произведенных на одной и той же аппаратной платформе.

7. Protic et al., *Distributed Shared Memory: Concepts and Systems.*

Это собрание из 28 опубликованных ранее статей представляет собой хорошее начало для знакомства с распределенной памятью совместного доступа. Здесь собрано множество классических статей по моделям, алгоритмам и вопросам реализации.

8. Stenstrom et al., «Trends in Shared Memory Multiprocessing».

В каком направлении развиваются мультипроцессоры? Авторы полагают, что будущее скорее за небольшими мультипроцессорами, чем за большими. Они также обсуждают модели, архитектуру и параллельное программное обеспечение.

9. Waldo, «Alive and Well: Jini Technology Today».

Эта статья представляет собой хорошую отправную точку для знакомства с системой Jini, ее компонентами и их интерфейсами. Возможно, иллюстрируя способ распространения информации в будущем, автор, вместо обычной библиографии, снабдил свою статью URL-ссылками на web-страницы.

Безопасность

1. *Computer*, Feb 2000.

Темой этого выпуска журнала *Computer* является биометрика, которой посвящены шесть журнальных статей. Они варьируются от введения в предмет до различных специфических технологий, вопросов конфиденциальности и юридических аспектов.

2. Denning, «The United States vs. Craig Neidorf».

Когда юный хакер обнаружил и опубликовал информацию о том, как работает телефонная система, ему было предъявлено обвинение в компьютерном мошенничестве. В статье описывается это судебное дело, затронувшее многие фундаментальные понятия, как, например, свобода слова. Следом за статьей несколько человек высказывают свои особые мнения, а Деннинг предоставляет контраргументы.

3. Denning, *Information Warfare and Security.*

Информация стала оружием, применяемым как в настоящей войне, так и в борьбе корпораций. Участники информационной войны не только стараются атаковать информационные системы противника, но также защитить собственные системы. В этой замечательной книге автор описывает каждый мыслимый аспект, относящийся к стратегии защиты и нападения, от фальсификации данных и до сетевого анализатора пакетов. Обязательное чтение для всех, кто серьезно интересуется вопросом компьютерной безопасности.

4. Hafner and Markoff, *Cyberpunk.*

Три захватывающие истории о молодых хакерах, вламывающихся в компьютеры по всему миру, рассказанные компьютерным обозревателем *New York Times* (Markoff). *Computer*, Feb 2000.

5. Johnson and Jajodia, «Exploring Steganography: Seeing the Unseen».

У стеганографии долгая история, начинающаяся в те дни, когда тайное послание писалось на выбритой голове посланника, после чего приходилось ждать, пока у посланника отрастут волосы. Сегодня применяются цифровые технологии, хотя зачастую они оказываются не менее сложными. Эта статья представляет собой хорошо написанное введение в данный предмет.

6. Ludwig, *The Giant Black Book of Computer Viruses, 2nd ed.*

Если вы хотите написать антивирусное программное обеспечение и вам необходимо понять, как работают вирусы, вплоть до самого нижнего уровня, эта книга для вас. В ней подробно обсуждается каждая разновидность вирусов, а для большинства из них также предоставляется фактический код на гибком диске. Однако для понимания книги обязательно требуется умение программировать на ассемблере Pentium.

7. Milojicic, «Security and Privacy».

Проблема безопасности имеет много граней, включающих операционные системы, сети, конфиденциальность и т. д. В данной статье публикуются интервью, взятые у шести экспертов в области безопасности.

8. Nachenberg, «Computer Virus-Antivirus Coevolution».

Как только разработчики антивирусного программного обеспечения находят способ обнаружения и нейтрализации некоторых классов компьютерных вирусов, создатели вирусов делают очередной шаг вперед, совершенствуют свои «произведения» и снова оказываются на шаг впереди. В данной статье обсуждаются различные аспекты этой игры в кошки-мышки. Автор не страдает чрезмерным оптимизмом и считает, что разработчики антивирусного программного обеспечения не могут выиграть эту войну, так что пользователей компьютеров утешить нечем.

9. Pfleeger, *Security in Computing, 2nd ed.*

Хотя по вопросу компьютерной безопасности написано много книг, в большинстве из них затрагивается только вопрос сетевой безопасности. В этой книге также обсуждается эта тема, но помимо этого три главы посвящены вопросу безопасности операционных систем. Таким образом, эта книга представляет собой хороший дополнительный источник информации по этой теме.

UNIX и Linux

1. Bovet and Cesati, *Understanding the Linux Kernel.*

Эта книга, вероятно, является самым лучшим изданием, содержащим всеобъемлющее обсуждение темы ядра операционной системы Linux. В ней описываются процессы, управление памятью, файловые системы, сигналы и многое другое.

2. IEEE, *Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language].*

Это стандарт. Некоторые разделы вполне удобочитаемы, особенно дополнение B, «Rationale and Notes» (обоснования и примечания), благодаря которому часто становится понятно, почему то или иное сделано именно так, а не иначе. Одно из преимуществ ссылки на стандарт заключается в том, что в этом документе по определению нет ошибок. Если какой-либо типографской ошибке в макросе удается преодолеть процесс редактирования, то после этого она уже не ошибка, а официальный факт.

3. Lewine, *POSIX Programmer's Guide.*

Эта книга описывает стандарт POSIX более подробно, чем документы стандарта, а также включает обсуждения с примерами темы преобразования старых программ в стандарт POSIX и методы разработки новых программ для среды POSIX.

4. Maxwell, *Linux Core Kernel Commentary.*

Первые 400 страниц этой книги содержат подмножество кода ядра операционной системы Linux. Последние 150 страниц представляют собой комментарий к коду, что соответствует стилю классической книги Джона Лайонза [211]. Если вы хотите понять ядро Linux во всех мельчайших деталях, эта книга поможет вам начать это непростое дело. Следует предупредить читателя: чтение 40 000 строк на языке С — занятие не для всех.

5. McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System.*

Заглавие книги говорит само за себя. Но на самом деле эта книга может также рассматриваться как общее руководство по UNIX вообще, так как внутреннее устрой-

ство различных версий системы UNIX весьма сходно. Это отличный источник информации для всех, кто хочет узнать о внутреннем устройстве системы UNIX.

Windows 2000

1. Cusumano and Selby, «How Microsoft Builds Software».

Вы никогда не задумывались о том, как вообще кому-то удалось написать программу, состоящую из 29 миллионов строк (как Windows 2000), да еще и заставить ее работать? Чтобы понять, как используется цикл создания и тестирования программ корпорации Microsoft для управления очень большими программными проектами, читайте эту статью.

2. Norton et al., *Complete Guide to Windows 2000 Professional.*

Если вы ищете книгу, обсуждающую вопросы настройки и использования Windows 2000, а также довольно подробно описывающую многие дополнительные особенности, такие как реестр, файловые системы FAT и NTFS, ActiveX, DCOM и работа в сети, то это правильный выбор. Эта книга занимает свое место между массой книг, сообщающих, где и что нажать, чтобы получить тот или иной эффект, и книгой Соломона и Руссиновича [309].

3. Rector and Newcomer, *Win32 Programming.*

Если вы ищете одну из тех 1500-страничных книг, в которых дается краткое изложение того, как писать программы для системы Windows, это не плохой экземпляр. Среди прочих тем, в ней описываются окна, устройства, графический вывод, ввод с клавиатуры и мыши, печать, управление памятью, библиотеки и синхронизация. Для чтения книги требуется знание языка программирования С или С++.

4. Solomon and Russinovich, *Inside Windows 2000, 3rd ed.*

Если вы хотите научиться работать в Windows 2000, для этого существуют сотни книг. Если же вы хотите узнать, как устроена операционная система Windows 2000, то эта книга окажется лучшим выбором. В ней описывается, и довольно подробно, множество внутренних алгоритмов и структур данных. Никакая другая книга не сравнится с этой.

Принципы проектирования

1. Brooks, «The Mythical Man Month: Essays on Software Engineering».

Фред Брукс был одним из разработчиков операционной системы OS/360. Он на собственном опыте научился отличать то, что работает, от того, что не работает. Советы, содержащиеся в этой остроумной, развлекательной и информативной книге, столь же ценны сегодня, как и четверть века назад, когда эта книга была написана.

2. Cooke et al., «UNIX and Beyond: An Interview with Ken Thompson».

Разработка операционных систем в значительной мере представляет собой искусство, нежели науку. Поэтому хороший способ узнать о предмете — слушать экспертов в этой области. Вряд ли можно найти лучшего эксперта, чем Кен Томпсон, который был одним из разработчиков систем UNIX, Inferno и Plan 9. В этом

пространном интервью Томпсон делится своими мыслями по поводу истории и перспектив данной области.

3. Corbato, «On Building Systems That Will Fail».

В своей лекции по поводу получения премии Тьюринга основатель системы разделения времени высказывает примерно те же соображения, что и Брукс в книге *Mythical Man-Month*. Он приходит к выводу, что все сложные системы в конце концов ждет крах и, чтобы иметь хоть какой-то шанс на успех, абсолютно необходимо избегать сложности и придерживаться простоты и элегантности проекта.

4. Crowley, *Operating Systems: A Design-Oriented Approach*.

В большинстве книг по операционным системам просто описываются основные понятия (процессы, виртуальная память и т. д.), а также приводится несколько примеров, но ничего не говорится о том, как проектировать операционные системы. Эта книга уникальна тем, что данной теме в ней посвящено четыре главы.

5. Lampson, «Hints for Computer System Design».

Батлер Лэмпсон, один из ведущих разработчиков передовых операционных систем в мире, за годы опыта собрал множество подсказок, предложений и руководящих указаний и поместил их в эту увлекательную и информативную статью. Как и книга Брукса, эта статья является обязательным чтением для каждого честолюбивого разработчика операционных систем.

6. Wirth, «A Plea for Lean Software».

Никлаус Вирт, знаменитый и опытный разработчик систем, выступает в данной статье в защиту простого и компактного программного обеспечения, основанного на нескольких простых понятиях, и против той разбухшей мешанины, какой является большая часть коммерческого программного обеспечения. Он доказывает свою точку зрения на примере собственной системы Oberon, представляющей собой ориентированную на работу в сети, основанную на графическом интерфейсе пользователя операционную систему, помещающуюся в 200 Кбайт, включая компилятор Oberon и текстовый редактор.

Алфавитный список литературы

1. Abdel-Hamid, T., and Madnick, s.: Software Project Dynamics: *An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.
2. Abram-Profeta, E. L., and Shin, K. G.: «Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers», *Proc. Int'l Conf. on Multimedia Comp. Syst.*, IEEE, pp. 66–75, 1998.
3. Abutaleb, A., and Li, V. O. K.: «Paging Strategy Optimization in Personal Communication Systems Wireless Networks», *Wireless Networks*, vol. 3, pp. 195–204, Aug. 1997.
4. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M.: «Mach: A New Kernel Foundation for UNIX Development», *Proc. Summer 1986 USENIX Conf.*, USENIX, pp. 93–112, 1986.

5. Adams, G. B. Iii, Agrawal, D. P., and Siegel, H. J.: «A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks», *Computer*, vol. 20, pp. 14–27, June 1987.
6. Adams, A., and Sasse, M. A.: «Taming the Wolf in Sheep's Clothing», *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 101–107, 1999.
7. Ahmad, I.: «Gigantic Clusters: Where Are They and What Are They Doing?» *IEEE Concurrency*, vol. 8, pp. 83–85, April–June 2000.
8. Alexandrov, A. D., Ibel, M., Schauser, K. E., and Scheiman, C. J.: «UFO: a Personal Global File System Based on User-Level Extensions to the Operating System», *Trans. on Computer Systems*, vol. 16, pp. 207–233, Aug. 1998.
9. Alfieri, R. A.: «An Efficient Kernel-Based Implementation of POSIX Threads», *Proc. Summer 1994 USENIX Tech. Conf.*, USENIX, pp. 59–72, June 1994.
10. Alvarez, G. A., Burkhard, W. A., and Cristian, F.: «Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering», *Proc. 24th Int'l Symp. on Computer Architecture*, ACM, pp. 62–72, 1997.
11. Anderson, T. E.: «The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors», *IEEE Trans. on Parallel and Distr. Systems*, vol. 1, pp. 6–16, Jan. 1990.
12. Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M.: «Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism», *ACM Trans. on Computer Systems*, vol. 10, pp. 53–79, Feb. 1992.
13. Andrews, G. R.: Concurrent Programming — Principles and Practice, Redwood City, CA: Benjamin/Cummings, 1991.
14. Andrews, G. R., and Schneider, F. B.: «Concepts and Notations for Concurrent Programming», *Computing Surveys*, vol. 15, pp. 3–43, March 1983.
15. Aron, M., and Druschel, P.: «Soft Timers: Efficient Microsecond Software Timer Support for Network Processing», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 223–246, 1999.
16. Arora, A. S., Blumofe, R. D., and Plaxton, C. G.: «Thread Scheduling for Multiprogrammed Multiprocessors», *Proc. Tenth Symp. on Parallel Algorithms and Architectures*, ACM, pp. 119–129, 1998.
17. Baker, F. T.: «Chief Programmer Team Management of Production Programming», *IBM Systems Journal*, vol. 11, pp. 1, 1972.
18. Baker-Harvey, M.: «ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems», *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 131–144, 1999.
19. Bala, K., Kaashoek, M. F., Weihl, W.: «Software Prefetching and Caching for Translation Lookaside Buffers», *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243–254, 1994.
20. Ballintijn, G., Van Steen, M., and Tanenbaum, A. S.: «Scalable Naming in Global Middleware», *Proc. 13th Int'l Conf. on Parallel and Distributed Systems*, ISCA, pp. 624–631, 2000.

21. Bays, C.: «A Comparison of Next-Fit, First-Fit, and Best-Fit», *Commun. of the ACM*, vol. 20, pp. 191–192, March 1977.

22. Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: *Linux Kernel Internals, 2nd ed.*, Reading, MA: Addison-Wesley, 1998.

23. Belady, L. A., Nelson, R. A., and Shedler, G. S.: «An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine», *Commun. of the ACM*, vol. 12, pp. 349–353, June 1969.

24. Bell, D., and La Padula, L.: «Secure Computer Systems: Mathematical Foundations and Model», Technical Report MTR 2547 v 2, Mitre Corp., Nov. 1973.

25. Ben-Ari, M.: *Principles of Concurrent Programming*, Upper Saddle River, NJ: Prentice Hall International, 1982.

26. Bernhardt, C., and Biersack, E. W.: «The Server Array: A Scalable Video Server Architecture», in *High-Speed Networking for Multimedia Applications*, Amsterdam: Kluwer Publishers, 1996.

27. Bershad, B. N., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., and Sirer, E. G.: «Protection is a Software Issue», *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 62–65, 1995a.

28. Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Chambers, C., and Eggers, S.: «Extensibility, Safety, and Performance in the SPIN Operating System», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 267–284, 1995b.

29. Bhoedjang, R. A. F.: Communication Architectures for Parallel-Programming Systems, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2000.

30. Bhoedjang, R. A. F., Ruhl, T., and Bal, H. E.: «User-Level Network Interface Protocols», *Computer*, vol. 31, pp. 53–60, Nov. 1998.

31. Bhuyan, L. N., Yang, Q., and Agrawal, D. P.: «Performance of Multiprocessor Interconnection Networks», *Computer*, vol. 22, pp. 25–37, Feb. 1989.

32. Biba, K.: «Integrity Considerations for Secure Computer Systems», Technical Report 76-371, U.S. Air Force Electronic Systems Division, 1977.

33. Birrell, A. D., and Nelson, B. J.: «Implementing Remote Procedure Calls», *ACM Trans. on Computer Systems*, vol. 2, pp. 39–59, Feb. 1984.

34. Bisdikian, C. C., and Patel, B. V.: «Issues on Movie Allocation in Distributed Video-on-Demand Systems», *Proc. Int'l Conf. on Commun.*, IEEE, pp. 250–255, 1995.

35. Blaum, M., Brady, J., Bruck, J., and Menon, J.: «EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures», *Proc. 21st Int'l Symp. on Computer Architecture*, ACM, pp. 245–254, 1994.

36. Blumofe, R. D., and Leiserson, C. E.: «Scheduling Multithreaded Computations by Work Stealing», *Proc. 35th Annual Symp. on Foundations of Computer Science*, IEEE, pp. 356–368, Nov. 1994.

37. Boehm, B. W.: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

38. Bolosky, W. J., Fitzgerald, R. P., and Douceur, J. R.: «Distributed Schedule Management in the Tiger Video Fileserver», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 212–223, 1997.

39. Born, G.,: *Inside the Microsoft Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.

40. Bovet, D. P., and Cesati, M.: *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2000.

41. Brandwein, R., Katseff, H., Markowitz, R., Mortenson, R., and Robinson, B.: «Nemesis: Multimedia Information Delivery», *Proc. Second ACM Int'l Conf. On Multimedia*, ACM, pp. 473–481, 1994.

42. Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M.: «A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility», *Proc. EurOpen Spring '91 Conf.*, EurOpen, pp. 13–32, 1991.

43. Brinch Hansen, P.: «The Programming Language Concurrent Pascal», *IEEE Trans. On Software Engineering*, vol. SE-1, pp. 199–207, June 1975.

44. Brooks, F. P., Jr.: The Mythical Man-Month: Essays on Software Engineering, Reading, MA: Addison-Wesley, 1975.

45. Brooks, F. P., Jr. «No Silver Bullet – Essence and Accident in Software Engineering», *Computer*, vol. 20, pp. 10–19, April 1987.

46. Brooks, F. P., Jr.: The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary edition, Reading, MA: Addison-Wesley, 1995.

47. Buchanan, M., and Chien, A.: «Coordinated Thread Scheduling for Workstation Clusters Under Windows NT», *The USENIX Windows NT Workshop*, USENIX, 1997.

48. Bugnion, E., Devine, S., Govil, K., and Rosenblum, M.: «Disco: Running Commodity Operating Systems on Scalable Multiprocessors», *Trans. on Computer Systems*, vol. 15, pp. 412–447, Nov. 1997.

49. Buzzard, G., Jacobson, D., Mackey, M., Marovich, S., and Wilkes, J.: «An Implementation of the Hamlyn Sender-Managed Interface Architecture», *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 245–259, Oct. 1996.

50. Cant, C.: *Writing Windows WDM Device Drivers*, Lawrence, KS: R&D Books, 1999.

51. Cao, P., Lin, S. B., Venkataraman, S., and Wilkes, J.: «The TickerTAIP Parallel RAID Architecture», *Trans. on Computer Systems*, vol. 12, pp. 236–269, Aug. 1994.

52. Cao, P., Felten, E. W., Karlin, A. R., and Li, K.: «A Study of Integrated Prefetching and Caching Strategies», *Proc. SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 188–197, 1995.

53. Carley, L. R., Ganger, G. R., and Nagle, D. F.: «MEMS-Based Integrated Circuit Mass Storage Systems», *Commun. of the ACM*, vol. 43. pp. 73–80, Nov. 2000.

54. Carr, R. W., and Hennessy, J. L.: «WSClock – A Simple and Effective Algorithm for Virtual Memory Management», *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp. 87–95, 1981.

55. Carriero, N., and Gelernter, D.: «The S/Net's Linda Kernel», *ACM Trans. On Computer Systems*, vol. 4, pp. 110–129, May 1986.

56. Carriero, N., and Gelernter, D.: «Linda in Context», *Commun. of the ACM*, vol. 32, pp. 444–458, April 1989.

57. Carter, J. B., Bennett, J. K., and Zwaenepoel, W.: «Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems», *Trans. on Computer Systems*, vol. 13, pp. 205–243, Aug. 1995.

58. Cerf, C., and Navasky, V.: *The Experts Speak*, New York: Random House, 1984.

59. Chandra, A., Adler, M., Goyal, P., and Shenoy, P.:s10 «Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors», *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 45–58, 2000.

60. Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D.: «Sharing and Protection in a Single-Address-Space Operating System», *Trans. on Computer Systems*, vol. 12, pp. 271–307, Nov. 1994.

61. Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A.: «RAID: High Performance Reliable Storage», *Comp. Surv.*, vol. 26, pp. 145–185, June 1994.

62. Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D.: «The Rio File Cache: Surviving Operating System Crashes», *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 74–83, 1996.

63. Chen, S., and Thapar, M.: «A Novel Video Layout Strategy for Near-Video-on-Demand Servers», *Proc. Int'l Conf. on Multimedia Computing and Systems*, IEEE, pp. 37–45, 1997.

64. Chen, S., and Towsley, D.: «A Performance Evaluation of RAID Architectures», *IEEE Trans, on Computers*, vol. 45, pp. 1116–1130, Oct. 1996.

65. Cheriton, D. R.: «An Experiment Using Registers for Fast Message-Based Interprocess Communication», *Operating Systems Review*, vol. 18, pp. 12–20, Oct. 1984.

66. Cheriton, D.: «The V Distributed System», *Commun. of the ACM*, vol. 31, pp. 314–333, March 1988.

67. Chervenak, A., Vellanki, V., and Kurmas, Z.: «Protecting File Systems: A Survey of Backup Techniques», *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.

68. Chow, T. C. K., and Abraham, J. A.: «Load Balancing in Distributed Systems», *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401–412, July 1982.

69. Clark, P. C., and Hoffman, L. J.: «BITS: A Smartcard Protected Operating System», *Commun. of the ACM*, vol. 37, pp. 66–70, Nov. 1994.

70. Coffman, E. G., Elphick, M. J., and Shoshani, A.: «System Deadlocks», *Computing Surveys*, vol. 3, pp. 67–78, June 1971.

71. Comer, D.: Operating System Design. The Xinu Approach, Upper Saddle River, NJ: Prentice Hall, 1984.
72. Cooke, D., Urban, J., and Hamilton, S.: «Unix and Beyond: An Interview with Ken Thompson», *Computer*, vol. 32, pp. 58–64, May 1999.
73. Corbalan, J., Martorell, X., and Labarta, J.: «Performance-Driven Processor Allocation», *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 59–71, 2000.
74. Corbato, F. J.: «On Building Systems That Will Fail», *Commun. of the ACM*, vol. 34, pp. 72–81, June 1991.
75. Corbato, F. J., Merwin-Daggett, M., and Daley, R. C.: «An Experimental Time-Sharing System», *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335–344, 1962.
76. Corbato, F. J., Saltzer, J. H., and Clingen, C. T.: «MULTICS – The First Seven Years», *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571–583, 1972.
77. Corbato, F. J., and Vyssotsky, V. A.: «Introduction and Overview of the MULTICS System», *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185–196, 1965.
78. Courtois, P. J., Heymans, F., and Parnas, D. L.: «Concurrent Control with Readers and Writers», *Commun. of the ACM*, vol. 10, pp. 667–668, Oct. 1971.
79. Cranor, C. D., and Parulkar, G. M.: «The UVM Virtual Memory System», *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 117–130, 1999.
80. Crowley, C:. Operating Systems: A Design-Oriented Approach, Chicago: Irwin, 1997.
81. Cusumano, M. A., and SELBY, R. W.: «How Microsoft Builds Software», *Commun. of the ACM*, vol. 40, pp. 53–61, June 1997.
82. Daley, R. C., and Dennis, J. B.: «Virtual Memory, Process, and Sharing in MULTICS», *Commun. of the ACM*, vol. 11, pp. 306–312, May 1968.
83. Dan, A., Sitaram, D., and Shahabuddin, P: «Scheduling Policies for an On-Demand Video Server with Batching», *Proc. Second Int'l Conf. on Multimedia*, ACM, pp. 15–23, 1994.
84. Dandamudi, S. P.: «Reducing Run Queue Contention in Shared Memory Multiprocessors», *Computer*, vol. 30, pp. 82–89, March 1997.
85. De Jonge, W., Kaashoek, M. F., and Hsieh, W. C.: «The Logical Disk: A New Approach to Improving File Systems», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 15–28, 1993.
86. Denning, D.: «A Lattice Model of Secure Information Flow», *Commun. of the ACM*, vol. 19, pp. 236–243, 1976.
87. Denning, D.: «The United States vs. Craig Neidorf», *Commun. of the ACM*, vol. 34, pp. 22–43, March 1991.
88. Denning, D.: *Information Warfare and Security*, Reading, MA: Addison-Wesley, 1999.
89. Denning, P. J.: «The Working Set Model for Program Behavior», *Commun. of the ACM*, vol. 11, pp. 323–333, 1968a.

90. Denning, P. J.: «Thrashing: Its Causes and Prevention», *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915–922, 1968b.

91. Denning, P. J.: «Virtual Memory», *Computing Surveys*, vol. 2, pp. 153–189, Sept. 1970.

92. Denning, P. J.: «Working Sets Past and Present», *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64–84, Jan. 1980.

93. Dennis, J. B., and VAN HORN, E. C.: «Programming Semantics for Multiprogrammed Computations», *Commun. of the ACM*, vol. 9, pp. 143–155, March 1966.

94. Devarakonda, M., Kish, B., and Mohindra, A.: «Recovery in the Calypso File System», *Trans. on Computer Systems*, vol. 14, pp. 287–310, Aug. 1996.

95. Diffie, W., and Hellman, M. E.: «New Directions in Cryptography», *IEEE Trans. On Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

96. Dijkstra, E. W.: «Cooperating Sequential Processes», in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.

97. Dijkstra, E. W.: «The Structure of THE Multiprogramming System», *Commun. of the ACM*, vol. 11, pp. 341–346, May 1968.

98. Douceur, J. R., and Bolosky, W. J.: «A Large-Scale Study of File-System Contents», *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 59–70, 1999.

99. Dubois, M., Scheurich, C., and Briggs, F. A.: «Synchronization, Coherence, and Event Ordering in Multiprocessors», *Computer*, vol. 21, pp. 9–21, Feb. 1988.

100. Druschel, P., Pai, V. S., and Zwaenepoel, W.: «Extensible Systems are Leading OS Research Astray», *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 38–42, 1997.

101. Duda, K. J., and Cheriton, D. R.: «Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 261–276, 1999.

102. Eager, D. L., Lazowska, E. D., and Zahorjan, J.: «Adaptive Load Sharing in Homogeneous Distributed Systems», *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 662–675, May 1986.

103. Eager, D. L., Vernon, M., and Zahorjan, J.: «Optimal and Efficient Merging Schedules for Video-on-Demand Servers», *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 199–202, 1999.

104. Edler, J., Lipkis, J., and Schonberg, E.: «Process Management for Highly Parallel UNIX Systems», *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp. 1–17, Sept. 1988.

105. Egan, J. I., and Teixeira, T. J.: *Writing a UNIX Device Driver, 2nd ed.*, New York: John Wiley, 1992.

106. El Gamal, A.: «A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms», *IEEE Trans. on Information Theory*, vol. IT-31, pp. 469–472, July 1985.

107. Ellis, C. S.: «The Case for Higher-Level Power Management», *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 162–167, 1999.

108. Engler, D. R., Gupta, S. K., and Kaashoek, M. F.: «AVM: Application-Level Virtual Memory», *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 72–77, 1995a.

109. Engler, D., Chelf, B., Chou, A., Hallem, S.: «Checking System Rules Using System-Specific Programmer-Written Compiler Extensions», *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1–16, 2000.

110. Engler, D. R., and Kaashoek, M. F.: «Exterminate All Operating System Abstractions», *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 78–83, 1995.

111. Engler, D. R., Kaashoek, M. F., and O'toole, J. Jr.: «Exokernel: An Operating System Architecture for Application-Level Resource Management», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 251–266, 1995.

112. Even, S.: *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.

113. Fabry, R. S.: «Capability-Based Addressing», *Commun. of the ACM*, vol. 17, pp. 403–412, July 1974.

114. Feeley, M. J., Morgan, W. E., Pighin, F. H., Karlin, A. R., Levy, H. M., and Thek-Kath, C. A.: «Implementing Global Memory Management in a Workstation Cluster», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 201–212, 1995.

115. Ferguson, D., Yemini, Y., and Nikolaou, C.: «Microeconomic Algorithms for Load Balancing in Distributed Computer Systems», *Proc. Eighth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 491–499, 1988.

116. Feustal, E. A.: «The Rice Research Computer — A Tagged Architecture», *Proc. AFIPS Conf.*, AFIPS, 1972.

117. Flinn, J., and Satyanarayanan, M.: «Energy-Aware Adaptation for Mobile Applications», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 48–63, 1999.

118. Fluckiger, F.: *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.

119. Ford, B., Hibler, M., Lepreau, J., Tullman, P., Back, G., Clawson, S.: «Micro-kernels Meet Recursive Virtual Machines», *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137–151, 1996.

120. Ford, B., and Susarla, S.: «CPU Inheritance Scheduling», *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 91–105, 1996.

121. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O.: «The Flux OSkit: A Substrate for Kernel and Language Research», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 38–51, 1997.

122. Fotheringham, J.: «Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store», *Commun. of the ACM*, vol. 4, pp. 435–436, Oct. 1961.

123. Gafsi, J., and Biersack, E. W.: «A Novel Replica Placement Strategy for Video Servers», *Proc. Sixth Int'l Workshop on Interactive and Distrib. Multimedia Systems*, ACM, pp. 321–335.

124. Geist, R., and Daniel, S.: «A Continuum of Disk Scheduling Algorithms», *ACM Trans. on Computer Systems*, vol. 5, pp. 77–92, Feb. 1987.

125. Gelernter, D.: «Generative Communication in Linda», *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 80–112, Jan. 1985.

126. Ghormley, D., Petrou, D., Rodrigues, S., Vahdat, A., and Anderson, T. E.: «SLIC: An Extensible System for Commodity Operating Systems», *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 39–46, 1998.

127. Gibson, G. A., and Van Meter, R.: «Network Attached Storage», *Commun. of the ACM*, vol. 43, pp. 37–45, Nov. 2000.

128. Gill, D. S., Zhou, S., and Sandhu, H. S.: «A Case Study of File System Workload in a Large-Scale Distributed Environment», *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 276–277, 1994.

129. Goldberg, L. A., Goldberg, P. W., Phillips,C. A., and SORKIN, G. B.: «Constructing Virus Phylogenies»,*Journal of Algorithms*, vol. 26, pp. 188–208, Jan. 1998.

130. Golden, D., and Pechura, M.: «The Structure of Microcomputer File Systems», *Commun. of the ACM*, vol. 29, pp. 222–230, March 1986.

131. Gong, L.: *Inside Java 2 Platform Security*, Reading, MA: Addison-Wesley, 1999.

132. Goodheart, B., and Cox, J.: *The Magic Garden Explained*, Upper Saddle River, NJ: Prentice Hall, 1994.

133. Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M.: «Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors», *Proc. 17th Symp. on Operating Systems Principles*, ACM, 1999, pp. 154–169 .

134. Goyal, P., Guo, X., and Vin, H. M.: «A Hierarchical CPU Scheduler for Multimedia Operating Systems», *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 107–121, 1996.

135. Graham, R.: «Use of High-Level Languages for System Programming», Project MAC Report TM-13, M.I.T., Sept. 1970.

136. Griffin, J. L., Schlosser, S. W., Ganger, G. R., and Nagle, D. F.: «Operating System Management of MEMS-based Storage Devices», *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87–102, 2000.

137. Grimm, R., and Bershad, B: «Security for Extensible Systems», *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 62–66, 1997

138. Griwodz, C., Bar. M., and Wolf, L. C.: «Long-Term Movie Popularity Models in Video-on-Demand Systems», *Proc. Fifth Int'l Conf. on Multimedia*, ACM, pp. 349–357, 1997.

139. Gropp, W., Lusk, E., and Skjellum, A.: Using MPI: Portable Parallel Programming *with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.

140. Grossman, D., and Silverman, H.: «Placement of Records on a Secondary Storage Device to Minimize Access Time»,*Journal of the ACM*, vol. 20, pp. 429–438, 1973

141. Hafner, K., and Markoff, J.: *Cyberpunk*, New York: Simon and Schuster, 1991.

142. Hand, S. M.: «Self-Paging in the Nemesis Operating System», *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 73–86, 1999.

143. Harbron, T. R.: *File Systems*, Upper Saddle River, NJ: Prentice Hall, 1988.

144. Harchol-Balter, M., and Downey, A. B.: «Exploiting Process Lifetime Distributions for Dynamic Load Balancing», *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 13–24, 1996.

145. Harrison, M. A., Ruzzo, W. L., and Ullman, J. D.: «Protection in Operating Systems», *Commun. of the ACM*, vol. 19, pp. 461–471, Aug. 1976.

146. Hart, J. M.: *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.

147. Hartig, H., Hohmuth, M., Liedtke, J., and Schonberg, S.: «The Performance of Kernel-Based Systems», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 66–77, 1997.

148. Hartman, J. H., and Ousterhout, J. K.: «The Zebra Striped Network File System», *Trans. on Computer Systems*, vol. 13, pp. 274–310, Aug. 1995.

149. Hauser, C., Jacobi, C., Theimer, M., Welch, B., and Weiser, M.: «Using Threads in Interactive Systems: A Case Study», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 94–105, 1993.

150. Havender, J. W.: «Avoiding Deadlock in Multitasking Systems», *IBM Systems Journal*, vol. 7, pp. 74–84, 1968.

151. Hebbard, B., et al.: «A Penetration Analysis of the Michigan Terminal System», *Operating Systems Review*, vol. 14, pp. 7–20, Jan. 1980.

152. Heidemann, J. S., and Popek, G. J.: «File-System Development with Stackable Layers», *Trans. on Computer Systems*, vol. 12, pp. 58–89, Feb. 1994.

153. Heybey, A., Sullivan, M., England, P.: «Calliope: A Distributed Scalable Multimedia Server», *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 75–86, 1996.

154. Hipson, P. D.: *Mastering Windows 2000 Registry*, Alameda, CA: Sybex, 2000.

155. Hoare, C. A. R.: «Monitors, An Operating System Structuring Concept», *Commun. of the ACM*, vol. 17, pp. 549–557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.

156. Holt, R. C.: «Some Deadlock Properties of Computer Systems», *Computing Surveys*, vol. 4, pp. 179–196, Sept. 1972.

157. Honeyman, P., Adamson, A., Coffman, K., Janakiraman, J., Jerdonek, R., and Rees, J.: «Secure Videoconferencing», *The Seventh USENIX Security Symp.*, USENIX, pp. 123–133, 1998.

158. Howard, J. H., Kazar, M. J., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and WEST, M. J.: «Scale and Performance in a Distributed File System», *ACM Trans. on Computer Systems*, vol. 6, pp. 55–81, Feb. 1988.

159. Huck, J., and Hays, J.: «Architectural Support for Translation Table Management in Large Address Space Machines», *Proc. 20th Int'l Symp. on Computer Architecture*, ACM, pp. 39–50, 1993.

160. Hutchinson, N. C., Manley, S., Federwisch, M., Harris, G., Hitz, D., Kleiman, S., and O'malley, S.: «Logical vs. Physical File System Backup», *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 239–249, 1999.

161. IEEE: Information Technology – Portable Operating System Interface (POSIX), Part 1: *System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, Inc., 1990

162. Isloor, S. S., and Marsland, T. A.: «The Deadlock Problem: An Overview», *Computer*, vol. 13, pp. 58–78, Sept. 1980.

163. Itzkovitz, A., and Schuster, A.: «MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs», *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 215–228, 1999.

164. Ivens, K.: *Optimizing the Windows Registry*, Foster City, CA: IDG Books Worldwide, 1998.

165. Johnson, K. L., Kaashoek, M.F., and Wallach, D. A.: «CRL: High-Performance All-Software Distributed Shared Memory», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213–226, 1995.

166. Johnson, N. F., and Jajodia, S.: «Exploring Steganography: Seeing the Unseen», *Computer*, vol. 31, pp. 26–34, Feb. 1998.

167. Jones, M. B., Rosu, D., and Rosu, M.-C.: «CPU Reservations and Time Constraints Efficient, Predictable Scheduling of Independent Activities», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 198–211, 1997.

168. Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceno, H., Hunt, R., Mazieres, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K.: «Application Performance and Flexibility on Exokernel Systems», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 52–65, 1997.

169. Kabay, M.: «Flashes from the Past», *Information Security*, p. 17, 1997.

170. Kallahalla, M., and Varman, P. J.: «Optimal Read-Once Parallel Disk Scheduling», *Proc. Sixth Workshop on I/O in Parallel and Distributed Systems*, ACM, pp. 68–77, 1999.

171. Karacali, B., Tai, K. C., and Vouk, M. A.: «Deadlock Detection of EFSMs Using Simultaneous Reachability Analysis», *Proc. Int'l Conference on Dependable Systems and Networks (DSN 2000)*, IEEE, pp. 315–324, 2000.

172. Karlin, A. R., Li, K., Manasse, M. S., and Owicki, S.: «Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor», *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 41–54, 1991.

173. Karlin, A. R., Manasse, M. S., Mcgeoch, L., and Owicki, S.: «Competitive Randomized Algorithms for Non-Uniform Problems», *Proc. First Annual ACM Symp. on Discrete Algorithms*, ACM, pp. 301–309, 1989.

174. Karpovich, J. F., Grimshaw, A. S., and French, J.C.: «Extensible File System (ELFS): An Object-Oriented Approach to High Performance File I/O», *Proc. Ninth Annual Conf. on Object-Oriented Programming Systems, Language, and Applications*, ACM, pp. 191–204, 1994.

175. Katcher, D. I., Kettler, K. A., and Strosnider, J. K.: «Real-Time Operating Systems for Multimedia Processing», *Proc. Hot Topics in Operating Systems V*, IEEE, 1995.

176. Kaufman, C., Perlman, R., and Speciner, M.: *Network Security*, Upper Saddle River, NJ: Prentice Hall, 1995.

177. Kavi, K., Browne, J. C., and Tripathi, A.: «Computer Systems Research: The Pressure is On», *Computer*, vol. 32, pp. 30–39, Jan 1999.

178. Keleher, P., Cox, A., Dwarkadas, S., and Zwaenepoel, W.: «TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems», *Proc. USENIX Winter 1994 Conf.*, USENIX, pp. 115–132, 1994.

179. Kernighan, B. W., and Pike, R.: *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.

180. Khalidi, Y. A., and Nelson, M. N.: «Extensible File Systems in Spring», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 1–14, 1993.

181. Klein, D. V.: «Foiling the Cracker: A Survey of, and Improvements to, Password Security», *Proc. UNIX Security Workshop II*, USENIX, Summer 1990.

182. Kleinrock, L.: *Queueing Systems. Vol. 1*, New York: John Wiley, 1975.

183. Kline, R. L., and Glinert, E. P.: «Improving GUI Accessibility for People with Low Vision», *Proc. Conf. on Human Factors in Computing Systems*, ACM, pp. 114–121, 1995.

184. Knuth, D. E.: The Art of Computer Programming, Volume 1: Fundamental Algorithms, *2nd Ed.*, Reading, MA: Addison-Wesley, 1973.

185. Kochan, S. G., and Wood, P. H.: *UNIX Shell Programming*, Indianapolis, IN: Hayden Books, 1990.

186. Kravets, R., and Krishnan, P.: «Power Management Techniques for Mobile Communication», *Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 157–168, 1998.

187. Krishnan, R.: «Timeshared Video-on-Demand: A Workable Solution», *IEEE Multimedia*, vol. 6, Jan.–March 1999, pp. 77–79.

188. Krueger, P., Lai, T.-H., and Dixit-Radiya, V. A.: «Job Scheduling is More Important than Processor Allocation for Hypercube Computers», *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, pp. 488–497, May 1994.

189. Kumar, V. P., and Reddy, S. M.: «Augmented Shuffle-Exchange Multistage Interconnection Networks», *Computer*, vol. 20, pp. 30–40, June 1987.

190. Kwok, Y.-K., Ahmad, I.: «Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors», *Computing Surveys*, vol. 31, pp. 406–471, Dec. 1999.

191. Lamport, L.: «Password Authentication with Insecure Communication», *Commun. of the ACM*, vol. 24, pp. 770–772, Nov. 1981.

192. Lampson, B. W.: «A Scheduling Philosophy for Multiprogramming Systems», *Commun. of the ACM*, vol. 11, pp. 347–360, May 1968.

193. Lampson, B. W.: «A Note on the Confinement Problem», *Commun. of the ACM*, vol. 10, pp. 613–615, Oct. 1973.

194. Lampson, B. W.: «Hints for Computer System Design», *IEEE Software*, vol. 1, pp. 11–28, Jan. 1984.

195. Lampson, B. W., and Sturgis, H. E.: «Crash Recovery in a Distributed Data Storage System», Xerox Palo Alto Research Center Technical Report, June 1979.

196. Landwehr, C. E.: «Formal Models of Computer Security», *Computing Surveys*, vol. 13, pp. 247–278, Sept. 1981.

197. Langendoen, K., Bhoedjang, R., and Bal, H. E.: «Models for Asynchronous Message Passing», *IEEE Concurrency*, vol. 5, pp. 28–37, April-June 1997.

198. Lebeck, A. R., Fan, X., Zeng, H., Ellis, C. S.: «Power Aware Page Allocation», Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ACM, 2000.

199. Lee, J. Y. B.: «Parallel Video Servers: A Tutorial», *IEEE Multimedia,* vol. 5, pp. 20–28, April-June 1998.

200. Lee, W., Su, D., Wijesekera, D., Srivastava, J., Kenchammana-Hosekote, D., and Foresti, M.: «Experimental Evaluation of PFS Continuous Media File System», *Proc. Sixth Int'l Conf. on Information and Knowledge Management,* ACM, pp. 246–253, 1997.

201. Leslie, I., Mcauley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fair-Bairns, R., and Hyden, E.: «The Design and Implementation of an Operating System to Support Distributed Multimedia Applications», *IEEE Journal on Selected Areas in Commun.*, vol. 14, pp. 1280–1297, July 1996.

202. Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., and Wulf, W. A.: «Policy/Mechanism Separation in Hydra», *Proc. Fifth Symp. on Operating Systems Principles*, ACM, pp. 132–140, 1975.

203. Lewine, D.: *POSIX Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.

204. Li, K.: «Shared Virtual Memory on Loosely Coupled Multiprocessors», Ph.D. Thesis, Yale Univ., 1986.

205. Li, K., and Hudak, P.: «Memory Coherence in Shared Virtual Memory Systems», *ACM Trans. on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

206. Li, K., Kumpf, R., Horton, P., and Anderson, T.: «A Quantitative Analysis of Disk Drive Power Management in Portable Computers», *Proc. 1994 Winter Conf.*, USENIX, pp. 279–291, 1994.

207. Liedtke, J.: «Improving IPC by Kernel Design», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 175–188, 1993.

208. Liedtke, J.: «On Micro-Kernel Construction», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 237–250, 1995.

209. Liedtke, J.: «Toward Real Microkernels», *Commun. of the ACM*, vol. 39, pp. 70–77, Sept. 1996.

210. Linde, R. R.: «Operating System Penetration», *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361–368, 1975.

211. Lions, J.: Lions' Commentary on Unix 6th Edition, with Source Code, San Jose, CA: Peer-to-Peer Communications, 1996.

212. Liu, C. L., and Layland, J. W.: «Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment», *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

213. Lo, V. M.: «Heuristic Algorithms for Task Assignment in Distributed Systems», *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 30–39, 1984.

214. Lorch, J. R., and Smith, A. J.: «Reducing Processor Power Consumption by Improving Processor Time Management In a Single-User Operating System», *Proc. Second Int'l Conf. on Mobile Computing and Networking*, ACM, pp. 143–154, 1996.

215. Lorch, J., and Smith, A. J.: «Apple Macintosh's Energy Consumption», *IEEE Micro*, vol. 18, pp. 54–63, Nov./Dec. 1998.

216. Lougher, P., Pegler, D., and Shepherd, D.: «Scalable Storage Servers for Digital Audio and Video», *Proc. IEE Int'l Conf. on Storage and Recording Systems*, London: IEE, pp. 140–143, 1994.

217. Lu, Y.-H., Simunic, T., De Micheli, G.: «Software Controlled Power Management», *Proc. Seventh Int'l Workshop on Hardware/Software Codesign*, ACM, pp. 157–161, 1999.

218. Ludwig, M. A.: The Giant Black Book of Computer Viruses, 2nd ed., Show Low, AZ: American Eagle Publications, 1998.

219. Lumb, C. R., Schindler, J., Ganger, G. R., and Nagle, D. F.: «Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives», *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87–102, 2000.

220. Maekawa, M., Oldehoeft, A.E., and Oldehoeft, R. R.: *Operating Systems: Advanced Concepts*, Menlo Park, CA: Benjamin/Cummings, 1987.

221. Malkewitz, R.: «Head Pointing and Speech Control as a hands-free interface to Desktop Computing», *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 182–188, 1998.

222. Manaris, B., and Harkreader, A.: «SUITEKeys: A Speech Understanding Interface for the Motor-Control Challenged», *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 108–115, 1998.

223. Mark, A. R.: «The Development of Destination-Specific Biometric Authentication», *Proc. Tenth Conf. on Computers, Freedom and Privacy*, ACM. pp. 77–80, 2000.

224. Markowitz, J. A.: «Voice Biometrics», *Commun. of the ACM*, vol. 43, pp. 66–73, Sept. 2000.

225. Marsh, B. D., Scott, M. L., Leblanc, T. J., and Markatos, E. P.: «First-Class User-Level Threads», *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 110–121, 1991.

226. Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., and Anderson, T. E.: «Improving the Performance of Log-Structured File Systems with Adaptive Methods», *Proc. 16th Symp. on Operating Systems Prin.*, ACM, pp. 238–251, 1997.

227. Maxwell, S. E.: *Linux Core Kernel Commentary*, Scottsdale, AZ: Coriolis, 1999.

228. Mazieres, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E.: «Separating Key Management from File System Security», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 124–139, 1999.

229. Mcdaniel, T.: «Magneto-Optical Data Storage», *Commun. of the ACM*, vol. 43, pp. 57–63, Nov. 2000.

230. Mckusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S.: *The Design and Implementation of the 4.4 BSD Operating System*, Reading, MA: Addison-Wesley, 1996.

231. Mckusick, M. J., JOY, W. N., LEFFLER, S. J., and FABRY, R. S.: «A Fast File System for UNIX», *ACM Trans. on Computer Systems*, vol. 2, pp. 181–197, Aug. 1984.

232. Medinets, D.: *UNIX Shell Programming Tools*, New York, NY: McGraw-Hill, 1999.

233. Mellor-Crummey, J. M., and SCOTT, M. L.: «Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors», *ACM Trans. on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

234. Mercer, C. W.: «Operating System Support for Multimedia Applications», *Proc. Second Int'l Conf on Multimedia*, ACM, pp. 492–493, 1994.

235. Miller, F. W.: «pk: A POSIX Threads Kernel», *FREENIX Track: USENIX Annual Technical Conference*, USENIX, pp. 179–182, 1999.

236. Milojicic, D.: «Operating Systems: Now and in the Future», *IEEE Concurrency*, vol. 7, pp. 12–21, Jan.–March 1999.

237. Milojicic, D.: «Security and Privacy», *IEEE Concurrency*, vol. 8, pp. 70–79, April–June 2000.

238. Monrose, F., and Rubin, A.: «Authentication Via Keystroke Dynamics», *Proc. Conf. on Computer and Communications Security*, ACM, pp. 48–56, 1997.

239. Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S., and Smith, F. D.: «Andrew: A Distributed Personal Computing Environment», *Commun. of the ACM*, vol. 29, pp. 184–201, March 1986.

240. Morris, R., and Thompson, K.: «Password Security: A Case History», *Commun. of the ACM*, vol. 22, pp. 594–597, Nov. 1979.

241. Mullender, S. J., and Tanenbaum, A. S.: «Immediate Files», *Software – Practice and Experience*, vol. 14, pp. 365–368, April 1984.

242. Myers, A. C., and Liskov, B.: «A Decentralized Model for Information Flow Control», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 129–142, 1997.

243. Nachenberg, C.: «Computer Virus-Antivirus Coevolution», *Commun. of the ACM*, vol. 40, pp. 46–51, Jan. 1997.

244. Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R.: *UNIX System Administration Handbook, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 2000.

245. Newham, C., and Rosenblatt, B.: *Learning the Bash Shell*, Sebastopol, CA: O'Reilly & Associates, 1998.

246. Newton, G.: «Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *Operating Systems Review*, vol. 13, pp. 33-44, April 1979.

247. Nieh, J., and Lam, M. S.: «The Design, Implementation and Evaluation of SMART a Scheduler for Multimedia Applications», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 184–197, 1997.

248. NIST (National Institute of Standards and Technology): FIPS Pub. 180–1, 1995.

249. Ng, S. W.: «Advances in Disk Technology: Performance Issues», *Computer*, vol. 31, pp. 75–81, May 1998.

250. Norton, P., Mueller, J., and Mansfield, R.: *Complete Guide to Windows 2000 Professional*, Indianapolis, IN: Sams, 2000.

251. Oki, B., Pfluegl, M., Siegel, A., and Skeen, D.: «The Information Bus – An Architecture for Extensible Distributed Systems», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 58–68, 1993.

252. Oney, W.: Programming the Microsoft Windows Driver Model, Redmond, WA: Microsoft Press, 1999.

253. Organick, E. I.: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.

254. Orlov, S. S.: «Volume Holographic Data Storage», *Commun. of the ACM*, vol. 43, pp. 47–54, Nov. 2000.

255. Ousterhout, J. K.: «Scheduling Techniques for Concurrent Systems», *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22–30, 1982.

256. Pai, V. S., Druschel, P., and Zwaenepoel, W.: «IO-Lite: A Unified I/O Buffering and Caching System», *Trans. on Computer Systems*, vol. 18, pp. 37-66, Feb. 2000.

257. Pakin, S., Karamcheti, V., Chien, A. A.: «Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors», *IEEE Concurrency*, vol. 5, pp. 60-73, April-June 1997.

258. Pankanti, S., Bolle, R. M., and Jain, A.: «Biometrics: The Future of Identification», *Computer*, vol. 33, pp. 46–49, Feb. 2000.

259. Pate, S. D.: *UNIX Internals A Practical Approach*, Reading, MA: Addison-Wesley, 1996.

260. Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J.: «Informed Prefetching and Caching», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 79–95, 1995.

261. Patterson, D. A., Gibson, G., and Katz, R.: «A Case for Redundant Arrays of Inexpensive Disks (RAID)», *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, ACM, pp. 109–166,1988.

262. Pentland, A., and Choudhury, T.: «Face Recognition for Smart Environments», *Computer*, vol. 33, pp. 50–55, Feb. 2000

263. Peterson, G. L.: «Myths about the Mutual Exclusion Problem», *Information Processing Letters*, vol. 12, pp. 115–116, June 1981.

264. Petrou, D., Milford, J., and Gibson, G.: «Implementing Lottery Scheduling», *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 1–14, 1999.

265. Petzold, C.: *Programming Windows, 5th ed.*, Redmond, WA: Microsoft Press, 1999.

266. Pfleeger, C. P.: *Security in Computing, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.

267. Philbin, J., Edler, J., Anshus, O. J., Douglas, C. C., and Li, K.: «Thread Scheduling for Cache Locality», *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 60–71, 1996.

268. Prechelt, L.: «An Empirical Comparison of Seven Programming Languages», *Computer*, vol. 33, pp. 23–29, Oct. 2000.

269. Protic, J., Tomasevic, M., and Milutinovic, V.: *Distributed Shared Memory: Concepts and Systems*, Los Alamitos, CA: IEEE Computer Society, 1998.

270. Rawson, F. L. III: «Experience with the Development of a Microkernel-Based, Multi-Server Operating System», *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 2–7, 1997.

271. Rector, B. E., and Newcomer, J. M.: *Win32 Programming*, Reading, MA: Addison-Wesley, 1997.

272. Reddy, A. L. N., and Wyllie, J. C.: «Disk Scheduling in a Multimedia I/O System», *Proc. ACM Multimedia Conf.*, ACM, pp. 225–233, 1992.

273. Reddy, A. L. N., and Wyllie, J. C.: «I/O Issues in a Multimedia System», *Computer*, vol. 27, pp. 69–74, March 1994.

274. Ritchie, D. M.: «Reflections on Software Research», *Commun. of the ACM*, vol. 27, pp. 758–760, Aug. 1984.

275. Ritchie, D. M., and Thompson, K.: «The UNIX Timesharing System», *Commun. of the ACM*, vol. 17, pp. 365–375, July 1974.

276. Rivest, R. L.: «The MD5 Message-Digest Algorithm», RFC 1320, April 1992.

277. Rivest, R. L., Shamir, A., and Adleman, L.: «On a Method for Obtaining Digital Signatures and Public Key Cryptosystems», *Commun. of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

278. Robbins, A.: UNIX in a Nutshell: A Desktop Quick Reference for SVR4 and Solaris 7, Sebastopol, CA: O'Reilly & Associates, 1999.

279. Rompogiannakis, Y., Nerjes, G., Muth, P., Paterakis, M., Triantafillou, P., and Weikum, G.: «Disk Scheduling for Mixed-Media Workloads in a Multimedia Server», *Proc. Sixth Int'l Conf. on Multimedia*, ACM, pp. 297–302, 1998.

280. Rosenblum, M., and Ousterhout, J. K.: «The Design and Implementation of a Log-Structured File System», *Proc. 13th Symp. on Oper. Sys. Prin.*, ACM, pp. 1–15, 1991.

281. Roselli, D., and Lorch, J. R.: «A Comparison of File System Workloads», *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 41-54, 2000.

282. Rozier, M., Abbrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Leonard, P., Langlois, S., and Neuhauser, W.: «Chorus Distributed Operating Systems», *Computing Systems*, vol. 1, pp. 305–379, Oct. 1988.

283. Ruemmler, C., and Wilkes, J.: «An Introduction to Disk Drive Modeling», *Computer*, vol. 27, pp. 17-28, March 1994.

284. Sackman, H., Erikson, W. J., and Grant, E. E.: «Exploratory Experimental Studies Comparing Online and Offline Programming Performance», *Commun. of the ACM*, vol. 11, pp. 3–11, Jan. 1968.

285. Saltzer, J. H.: «Protection and Control of Information Sharing in MULTICS», *Commun. of the ACM*, vol. 17, pp. 388–402, July 1974.

286. Saltzer, J. H., Reed, D. P., and Clark, D. D.: «End-to-End Arguments in System Design», *Trans. on Computer Systems*, vol. 2, pp. 277–277, Nov. 1984.

287. Saltzer, J. H., and Schroeder, M. D.: «The Protection of Information in Computer Systems», *Proc. IEEE*, vol. 63, pp. 1278–1308, Sept. 1975.

288. Salus, P. H.: «UNIX At 25», *Byte*, vol. 19, pp. 75–82, Oct. 1994.

289. Sandhu, R. S.: «Lattice-Based Access Control Models», *Computer*, vol. 26, pp. 9–19, Nov. 1993.

290. Santry, D., Feeley, M. J., Hutchinson, N. C., Veitch, A. C.: «Elephant: The File System That Never Forgets», *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 2–7, 1999a.

291. Santry, D., Feeley, M., Hutchinson, N., Veitch, A. C., Carton, R. W., and Ofir, J.: «Deciding When to Forget in the Elephant File System», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 110–123, 1999b.

292. Satyanarayanan, M., Howard, J. H., Nichols, D. N., Sidebotham, R. N., Spector, A. Z., and West, M. J.: «The ITC Distributed File System: Principles and Design», *Proc. of the Tenth Symp. on Operating System Prin.*, ACM, pp. 35–50, 1985.

293. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T.: «Eraser: A Dynamic Data Race Detector for Multithreaded Programs», *Trans. on Computer Systems*, vol. 15, pp. 391–411, Nov. 1997.

294. Scales, D. J., and Gharachorloo, K.: «Towards Transparent and Efficient Software Distributed Shared Memory», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 157–169, 1997.

295. Schmidt, B. K., Lam, M. S., and Northcutt, J. D.: «The Interactive Performance of Slim: A Stateless, Thin-Client Architecture», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 32–47, 1999.

296. Scott, M., Leblanc, T., and Marsh, B.: «Multi-model Parallel Programming in Psyche», *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp. 70–78, 1990.

297. Seawright, L. H., and Mackinnon, R. A.: «VM/370 — A Study of Multiplicity and Usefulness», *IBM Systems Journal*, vol. 18, pp. 4–17, 1979.

298. Segarra, M.-T., and Andri, F.: «MFS: a Mobile File System Using Generic System Services», *Proc. 1999 Symp. on Applied Computing*, ACM, pp. 419–420, 1999.

299. Seltzer, M., Endo, Y., Small, C., and Smith, K.: «Dealing with Disaster: Surviving Misbehaved Kernel Extensionsm». *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 213–227, 1994.

300. Shenoy, P. J., and Vin, H. M.: «Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers», *Perf. Eval. Journal*, vol. 38, pp. 175–199, 1999.

301. Shenoy, P. J., Goyal, P., and Vin, H. M.: «Architectural Considerations for Next Generation File Systems», *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 457–467, 1999.

302. Silberschatz, A., Galvin, P. B., and Gagne, G: *Applied Operating System Concepts*, New York: Wiley, 2000.

303. Simon, R. J.: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.

304. Sitaram, D., and Dan, A.: *Multimedia Servers*, San Francisco, CA: Morgan Kaufman, 2000.

305. Slaughter, L., Oard, D. W., Warnick, V. L., Harding, J. L., and Wilkerson, G. J.: «A Graphical Interface for Speech-Based Retrieval», *Proc. Third ACM Conf. on Digital Libraries,* ACM, pp. 305–306, 1998.

306. Small, C., and Seltzer, M.: «MiSFIT: constructing Safe Extensible Systems», *IEEE Concurrency*, vol. 6, pp. 34–41, July-Sept. 1998.

307. Smith, D. K., and Alexander, R. C.: Fumbling the Future: How Xerox Invented, Then *Ignored, the First Personal Computer*, New York: William Morrow, 1988.

308. Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.

309. Solomon, D. A., and Russinovich, M.E.: *Inside Windows 2000, 3rd ed.*, Redmond, WA: Microsoft Press, 2000

310. Spafford, E., Heaphy, K., and Ferbrache, D.: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.

311. Stallings, W.: *Operating Systems, 4th Ed.*, Upper Saddle River, NJ: Prentice Hall, 2001.

312. Steenkiste, P. A.: «A Systematic Approach to Host Interface Design for High-Speed Networks», *Computer*, vol. 27, pp. 47-57, March 1994.

313. Steinmetz, R., and Nahrstedt, K.: Multimedia: Computing, Communications and *Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.

314. Stenstrom, P., Hagersten, E., Lilja, D. J., Martonosi, M., and Venugopal, M.: «Trends in Shared Memory Multiprocessing», *Computer*, vol. 30, pp. 44–50, 1997.

315. Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S., and Scott, M.: «Cashmere — 2L Software Coherent Shared Memory on a Clustered Remote-Write Network», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170–183, 1997.

316. Stevens, W. R.: Advanced Programming in the UNIX Environment, Reading, MA: Addison-Wesley, 1992.

317. Stoll, C.: The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage, New York: Doubleday, 1989.

318. Stone, H. S., and Bokhari, S. H.: «Control of Distributed Processes», *Computer*, vol. 11, pp. 97–106, July 1978.

319. Tai, K.C., and Carver, R. H.: «VP: A New Operation for Semaphores», *Operating Systems Review*, vol. 30, pp. 5–11, July 1996.

320. Talluri, M., and Hill, M. D.: «Surpassing The Tlb Performance of Superpages With Less Operating System Support», *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 171–182, 1994.

321. Talluri, M., Hill, M. D., and Khalidi, Y. A.: «A New Page Table for 64-bit Address Spaces», *Proc. 15th Symp. on Operating Systems Prin.*, ACM, pp. 184–200, 1995.

322. Tanenbaum, A. S.: Operating Systems: Design and Implementation, Upper Saddle River, NJ: Prentice Hall, 1987.

323. Tanenbaum, A. S.: *Computer Networks*, Upper Saddle River, NJ: Prentice Hall, 1996.

324. Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J., and Van Rossum, G.: «Experiences with the Amoeba Distributed Operating System», *Commun. of the ACM*, vol. 33, pp. 46–63, Dec. 1990.

325. Tanenbaum, A. S., and Van Steen, M. R.: *Distributed Systems*, Upper Saddle River, NJ: Prentice Hall, 2002.

326. Tanenbaum, A. S., and Woodhull, A. S.: Operating Systems: Design and Implementation, *2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.

327. Taylor, R. N, Medvidovic, N., Anderson, K. M., Whitehead, E. J., and Robbins, J. E.: «A Component- and Message-Based Architectural Style for GUI Software», *Proc. 17th Int'l Conf. on Software Engineering*, ACM, pp. 295–304, 1995.

328. Teory, T. J.: «Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems», *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1–11, 1972.

329. Thekkath, C. A., Mann, T., and Lee, E. K.: «Frangipani: A Scalable Distributed File System», *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 224–237, 1997.

330. Thompson, K.: «Reflections on Trusting Trust», *Commun. of the ACM*, vol. 27, pp. 761–763, Aug. 1984.

331. Trono, J. A.: «Comments on Tagged Semaphores», *Operating Systems Review*, vol. 34, pp. 7–11, Oct. 2000.

332. Tucker, A., and Gupta, A.: «Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors», *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp. 159–166, 1989.

333. Uhlig, R., Nagle, D., Stanley, T., Mudge, T., Secrest, S., and Brown, R.: «Design Tradeoffs for Software-Managed TLBs», *ACM Trans. on Computer Systems*, vol. 12, pp. 175–205, Aug. 1994.

334. Vahalia, U.: *UNIX Internals — The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 1996.

335. Van Buskirk, R., and Lalomia, M.: «A Comparison of Speech and Mouse/Keyboard GUI Navigation», *Conference Companion on Human Factors in Computing Systems*, ACM, p. 96, 1995.

336. Van Doorn, L.: The Design and Application of an Extensible Operating System, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2001.

337. Van Doorn, L., Homburg, P., and Tanenbaum, A. S.: «Paramecium: An Extensible Object-Based Kernel», *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 86–89, 1995.

338. Van Steen, M., Hauck, F. J., Ballintijn, G., and Tanenbaum, A. S.: «Algorithmic Design of the Globe Wide-Area Location Service», *Computer Journal*, vol. 41, pp. 207–310, 1998a.

339. Van Steen, M., Hauck, F. J., Homburg, P., and Tanenbaum, A. S.: «Locating Objects in Wide-Area Systems», *IEEE Communications Magazine*, vol. 36, pp. 104–109, Jan. 1998b.

340. Van Steen, M., Homburg, P., and Tanenbaum, A. S.: «Globe: A Wide-Area Distributed System», *IEEE Concurrency*, vol. 7, pp. 70–78, Jan.–March 1999a.

341. Van Steen, M., Tanenbaum, A. S., Kuz, I., and Sips, H. J.: «A Scalable Middleware Solution for Advanced Wide-Area Web Services», *Distributed Systems Engineering*, vol. 7, pp. 34–42, 1999b.

342. Vaswani, R., and Zahorjan, J.: «The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors», *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 26–40, 1991.

343. Venkatasubramanian, N., and Ramanathan, S: «Load Management in Distributed Video Servers», *Proc. 17th Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 528–535, 1997.

344. Vinoski, S.: «CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments», *IEEE Communications Magazine*, vol. 35, pp. 46–56, Feb 1997.

345. Viscarola, P. G., and MASON, W. A.: *Windows NT Device Driver Development*, Indianapolis, IN: Macmillan Technical Publishing, 1999.

346. Vogels, W.: «File System Usage in Windows NT 4.0», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 93–109, 1999.

347. Von Eicken, T., Basu, A., Buch, V., and Vogels, W.: «U-Net: a User-Level Network Interface for Parallel and Distributed Computing», *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 40–53, 1995.

348. Von Eicken, T., Culler, D., Goldstein, S. C., Schauser, K. E.: «Active Messages: A Mechanism for Integrated Communication and Computation», *Proc. 19th Int'l Symp. on Computer Architecture*, ACM, pp. 256–266, 1992.

349. Wahbe, R., Lucco, S., Anderson, T., and Graham, S.: «Efficient Software-Based Fault Isolation», *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 203–216, 1993.

350. Waldo, J.: «The Jini Architecture for Network-Centric Computing», *Commun. of the ACM*, vol. 42, pp. 76–82, July 1999.

351. Waldo, J.: «Alive and Well: Jini Technology Today», *Computer*, vol. 33, pp. 107-109, June 2000.

352. «Lottery Scheduling: Flexible Proportional-Share Resource Management», *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1–12, 1994.

353. Walker, W., and Cragon, H. G.: «Interrupt Processing in Concurrent Processors», *Computer*, vol. 28, pp. 36–46, June 1995.

354. Wan, G., and Lin, E.: «A Dynamic Paging Scheme for Wireless Communication Systems», *Proc. Third Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 195–203, 1997

355. Wang, C., Goebel, V., and Plagemann, T.: «Techniques to Increase Disk Access Locality in the Minorca Multimedia File System», *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 147–150, 1999.

356. Wang, R. Y., Anderson, T. E., and Patterson, D. A.: «Virtual Log Based File Systems for a Programmable Disk», *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 29–43, 1999.

357. Weiser, M., Welch, B., Demers, A., and Shenker, S.: «Scheduling for Reduced CPU Energy», *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 13–23, 1994.

358. Wilkes, J., Golding, R., Staelin, C, and Sullivan, T.: «The HP AutoRAID Hierarchical Storage System», *ACM Trans. on Computer Systems*, vol. 14, pp. 108–136, Feb. 1996.

359. Wirth, N.: «A Plea for Lean Software», *Computer*, vol. 28, pp. 64–68, Feb. 1995.

360. Wolman, A., Voelker, M., Sharma, N., Cardwell, N., Karlin, A., and Levy, H. M.: «On the Scale and Performance of Cooperative Web Proxy Caching», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 16–31, 1999.

361. Wong, C. K.: Algorithmic Studies in Mass Storage Systems, New York: Computer Science Press, 1983.

362. Wong, P. C., and Lee, Y. B.: «Redundant Array of Inexpensive Servers (RAIS)», *Proc. Int'l Conf. on Commun.*, IEEE, pp. 787–792, 1997.

363. Worthington, B. L., Ganger, G. R., and Patt, Y. N.: «Scheduling Algorithms for Modern Disk Drives», *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, pp. 241–251, 1994.

364. Wu, M., and Shu, W.: «Scheduling for Large-Scale Parallel Video Servers», *Proc. Sixth Symp. on Frontiers of Massively Parallel Computation*, IEEE, pp. 126–133, 1996.

365. Wulf, W. A., Cohen, E. S., Corwin, W. M., Jones, A. K., Levin, R., Pierson, C., and Pollack, F. J.: «HYDRA: The Kernel of a Multiprocessor Operating System», *Commun. of the ACM*, vol. 17, pp. 337–345, June 1974.

366. Young, M., Tevanian, A., Jr., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R.: «The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System», *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63–76, 1987.

367. Zachary, G. P.: *Showstopper*, New York: Maxwell MacMillan, 1994.

368. Zahorjan, J., Lazowska, E. D., and Eager, D. L.: «The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems», *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 180–198, April 1991.

369. Zekauskas, M. J., Sawdon, W. A., and Bershad, B. N.: «Software Write Detection for a Distributed Shared Memory», *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87–100, 1994.

370. Zobel, D.: «The Deadlock Problem: A Classifying Bibliography», *Operating Systems Review*, vol. 17, pp. 6–16, Oct. 1983.

371. Zuberi, K. M., Pillai, P., and Shin, K. G.: «EMERALDS: A Small-Memory Real-Time Microkernel», *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 277–299, 1999.

372. Zwicky, E. D.: «Torture-Testing Backup and Archive Programs: Things You Ought to Know but Probably Would Rather Not», *Proc. Fifth Conf. on Large Installation Systems Admin.*, USENIX, pp. 181–190, 1991.

Алфавитный указатель

Е

Ж

З

И

К

Л

М

Р

С

Э

Я

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

Башкортостан
Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа), маг. «Оазис», ул. Чернышевского, д. 88, тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток
Владивосток, «Приморский торговый дом книги», тел./факс (4232) 23-82-12. Почтовый адрес: 690091, Владивосток, ул. Светланская, д. 43.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс», тел. (4212) 30-54-47, факс 22-73-30. Почтовый адрес: 680000, Хабаровск, ул. Ким-Ю-Чена, д. 21.
E-mail: postmaster@bookmirs.khv.ru

Хабаровск, «Книжный мир», тел. (4212) 32-85-51, факс 32-82-50. Почтовый адрес: 680000, Хабаровск, ул. Карла Маркса, д. 37.
E-mail: postmaster@worldbooks.knt.ru

Европейские регионы России
Архангельск, «Дом книги», тел. (8182) 65-41-34, факс 65-41-34. Почтовый адрес: 163061, пл. Ленина, д. 3. E-mail: book@atnet.ru

Калининград, «Вестер», тел./факс (0112) 21-56-28, 21-62-07. Почтовый адрес: 236040, Калининград, ул. Победы, д. 6. Магазин «Книги & книжечки».
E-mail: nshibkova@vester.ru;
http://www.vester.ru

Ростов-на-Дону, ПБОЮЛ Остроменский, пр. Соколова, д. 73, тел./факс (8632) 32-18-20.
E-mail: ostrom@don.sitek.net

Северный Кавказ
Ессентуки, «Россы», ул. Октябрьская, 424, тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь
Иркутск, «ПродаЛитъ», тел. (3952) 59-13-70, факс 51-30-70. Почтовый адрес: 664031, Иркутск, ул. Байкальская, д. 172, а/я 1397.
E-mail: prodalit@irk.ru; http:/.www.prodalit.irk.ru

Иркутск, «Антей-книга», тел./факс (3952) 33-42-47. Почтовый адрес: 664003, Иркутск, ул. Карла Маркса, д. 20. E-mail: antey@irk.ru

Красноярск, «Книжный мир», тел./факс (3912) 27-39-71. Почтовый адрес: 660049, Красноярск, пр. Мира, д. 86. E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги», тел. (3466) 23-27-14, факс 23-59-50. Почтовый адрес: 628606, Нижневартовск, пр. Победы, д.12.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга», тел. (3832) 36-10-26, факс 36-10-27. Почтовый адрес: 630117, Новосибирск, а/я 560.
E-mail: office@top-kniga.ru; http://www.top-kniga.ru

Тюмень, «Друг», тел./факс (3452) 21-34-39, 21-34-82. Почтовый адрес: 625019, ул. Республики, д. 211. E-mail: drug@tyumen.ru

Тюмень, «Фолиант», тел. (3452) 27-36-06, факс 27-36-11. Почтовый адрес: 625039, Тюмень, ул. Харьковская, д. 83а. E-mail: foliant@tyumen.ru

Татарстан
Казань, «Таис», тел. (8432) 72-34-55, факс 72-27-82. Почтовый адрес: 420073, Казань, ул. Гвардейская, д. 9а. E-mail: tais@bancorp.ru

Урал
Екатеринбург, магазин № 14, ул. Челюскинцев, д. 23, тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга», ул. Ключевская, д. 5, тел. (3432) 42-07-75, факс 42-56-00.
E-mail: valeo@etel.ru

Э. ТАНЕНБАУМ

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Это с нетерпением ожидаемое, переработанное и исправленное издание всемирного бестселлера включает в себя сведения о последних достижениях в области технологий операционных систем. Книга построена на примерах и содержит информацию, необходимую для понимания функционирования современных операционных систем.

Благодаря практическому опыту, приобретенному при разработке нескольких операционных систем, и высокому уровню знания предмета Эндрю Таненбаум смог ясно и увлеченно рассказать о сложных вещах. В книге приводится множество важных подробностей, которых нет ни в одном другом издании.

Особое внимание в книге уделяется:

- компьютерной безопасности, надежности, мультимедийным операционным системам и многопроцессорным системам;
- графическим интерфейсам пользователя, многопроцессорным операционным системам, сетевым терминалам, файловым системам компакт-дисков, вирусам;
- управлению энергопотреблением на лэптопах, системам RAID, мягким таймерам, стабильным хранилищам, справедливому и трехуровневому планированию, новым алгоритмам замещения страниц.

ISBN 5-318-00299-4

9 785318 002991

ПИТЕР®
WWW.PITER.COM

PH PTR